Tryton Test

Release 0.1

May 21, 2020

Contents

Welcome to Tryton Documentation. Here you'll get the resources you need to set up and use Tryton for your business.

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-tier architecture consists of the Tryton client, the Tryton server, and the Database management system (mainly PostgreSQL).

It is business software, ideal for companies of any size, easy to use, complete and 100% Open Source. In this documentation, we'll be walking you through on the best way to set up Tryton to start your business. For more information join our Discussion-group

Note: Select the area that best describes your need.

CHAPTER 1

Deskop Client-User

CHAPTER 2

Web Client-User

CHAPTER 3

System Administrator

CHAPTER 4

Developer

Backend

GIS Adds GIS support.

4.1 Tryton Desktop Client

Tryton is a Graphical User Interface to Tryton based on GTK and Python.

The **Desktop Client** are the desktop application users or the end users who accesses the application locally. In this documentation, we'll teach you how to set up **Tryton Client** to boost your business. Below is a list of what we'll cover.

4.1.1 Contents

Installing tryton

Installation

Once you've downloaded and unpacked a tryton source release, enter the directory where the archive was unpacked, and run:

```
python setup.py install
```

Note that you may need administrator/root privileges for this step, as this command will by default attempt to install tryton to the Python site-packages directory on your system.

For advanced options, please refer to the easy_install and/or the distutils documentation:

To use without installation, run `bin/tryton` from where the archive was unpacked.

Client Usage

This document is the reference about the concepts of the graphical user interface (also known as *Tryton client*) of the Tryton application framework.

Name

tryton - Graphical user client of the Tryton application framework

Synopsis

```
tryton [options] [url]
```

On startup the login dialog is displayed.

Options

–version	Show program version number and exit
-h, –help	Show help message and exit
-c *FILE,* –config=FILE	Specify alternate *configuration file*
-d, –dev	Enable development mode, which deactivates client side caching
-v, –verbose	Enable basic debugging
-l *LOG_LEVEL,* –log-level=LOG_LEVEL	Specify the log level: DEBUG, INFO, WARNING, ERROR, CRITICAL
-u *LOGIN,* –user=LOGIN	Specify the login user
-s *SERVER,* –server=SERVER	Specify the server hostname:port

URL

When an url is passed, the client will try to find already running client that could handle it and send to this one to open the url. If it doesn't find one then it will start the GUI and open the url itself.

The url schemes are:

tryton://<hostname>[:<port>]/<database>/model/<model name>[/<id>][;parameters]

tryton://<hostname>[:<port>]/<database>/wizard/<wizard name>[;parameters]

tryton://<hostname>[:<port>]/<database>/report/<report name>[;parameters]

where *parameters* are the corresponding fields of actions encoded in JSON.

Note: *model* is for *act_window*

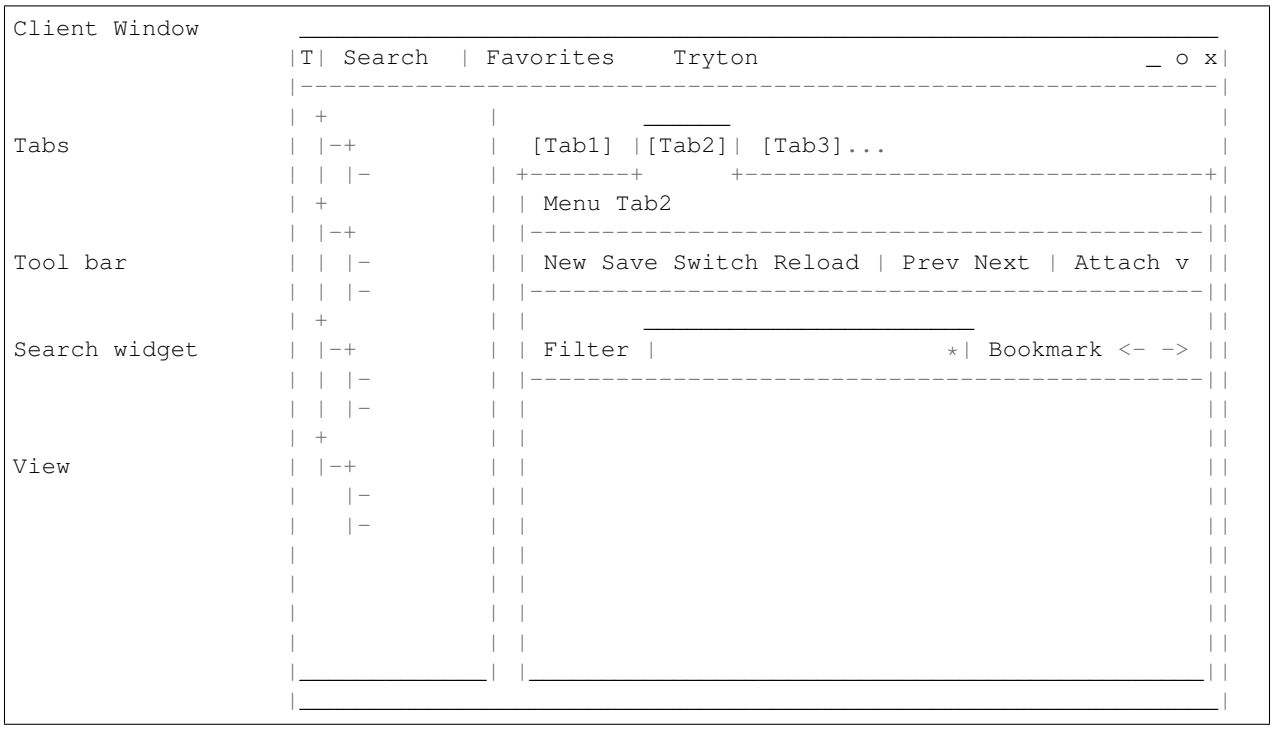
Note: *report* must have at least a data parameter with *ids*, *id* and *model name*

Overview

On startup the login dialog is displayed. It allows to select a existing profile (or to manage them) or to enter the host and database information.

The following schematic illustration of the Tryton client shows the names of all important visual parts.

Figure: Tryton client application:



Tabbed Main Frame

This part of the client contains all the related contents and functions provided by the *Tryton Server Modules*. All aspects inside the *main frame* depend at least on the individual set of installed modules.

The main frame provides a tabbed document interface to arrange different views side by side. New *Tabs* are opened by special *Actions*, like choosing a menu item or clicking some action buttons. All tabs include titles which show the name of the provided view.

Tabs can be arranged by Drag and Drop.

Note: Inside *Views* there can be tabs, too.

Menu

The *menu* does not contain fixed menu items. All of them are dynamically provided by the actual set of the installed *Modules* depending on the access rules of the current user. If a menu item is clicked, the appropriate action will open in a new tab.

A search field allows to quickly filter the menu items by name and to search in models for which the global search is enabled.

Application Menu

The following section describes the action of the application menu. A rule of thumb: All items of the menu bar that are suffixed by three dots (...) will open an intermediate *Dialog* for setting up the provided menu action. Most dialog provide a *Cancel* button, used to stop the complete dialog process.

Preferences:

> A preference dialog opens, where the actual user can show and edit his personal settings. All user preferences are stored server side. I.e. logging in with the same credentials from different computers always restores the same preferences.

Options

The Options menu sets up several visual and context depending preferences.

Toolbar:

- Default: Shows labels and icons as defaulted in the GTK configuration.
- Text and Icons: Shows labels and icons in the tool bar.
- Icons: Shows icons only in the tool bar.
- Text: Shows labels only in the tool bar.

Form:

- Save Column Width: Check box to enable saving of manually adjusted widths of columns in lists and trees.
- Save Tree Expanded State: Check box to enable saving of expanded and selected nodes in trees/lists.
- Spell Checking: Check box to enable spell checking in fields.

PDA Mode:

> When activated, the client display in a condensed mode.

Search Limit:

> Open a dialog to set up the maximum number of records displayed on a list.

Email:

> Open a dialog to set up an email reader.
>
> - Command Line: The command line calling the email reader.
> - Placeholders:
>   - `${to}`: the destination email address
>   - `${cc}`: the carbon copy email address
>   - `${subject}`: the subject of the email
>   - `${body}`: the body of the email
>   - `${attachment}`: the attachment of the email
> - Examples:

- Thunderbird 2 on Linux: `thunderbird -compose "to='${to}',cc='${cc}',subject='${subject}',body='${body}',attachment='file://${attachment}'"`
- Thunderbird 2 on Windows XP SP3: `"C:\\Program Files\\Mozilla Thunderbird\\thunderbird.exe" -compose to="${to}",cc="${cc}",subject="${subject}",body="${body}",attachment="${attachment}"`

Note: The path of *Program Files* may vary dependent on the localization of your Windows version.

Check Version:

Check box to enable the check of new bug-fix version.

Help

Keyboard Shortcuts...:

Shows the information dialog of the predefined keyboard shortcut map.

- Edition Widgets: Shows shortcuts working on text entries, relation entries and date/time entries.

About...:

License, Contributors, Authors of Tryton

Tool Bar

The tool bar contains the functionalities linked to the current tab. Some operations are working with one record or with a selection of *Records*. In *Form View* the actual record is selected for operations. In *Tree View* all selected records are used for operations.

New:

Creates a new record.

Save:

Saves the actual record.

Switch View:

Switches the actual view aspect to:

- *Form View*
- *Tree View*
- *Graph View*

Not all views provide all aspects.

Reload/Undo:

Reloads the content of the actual tab. Undoes changes, if save request for the current record is denied.

Duplicate:

Duplicates the content of the actual record in a newly created record.

Delete:

Deletes the selected or actual record.

Previous:

Goes to the last record in a list (sequence).

Next:

Goes to the next record in a list (sequence).

Search:

Goes to the search widget.

View Logs...:

Shows generic information of the current record.

Show revisions...:

Reload the current view/record at a specific revision.

Close Tab:

Closes the current tab. A Request *Dialog* opens in case of unsaved changes.

Attachment:

The attachment item handles the document management system of Tryton which is able to attach files to any arbitrary *Model*. On click it opens the attachments *Dialog*. The default dialog shows a list view of the attached files and links.

Actions...:

Shows all actions for the actual view, model and record.

Relate...:

Shows all relate view for the actual view, model and record.

Report...:

Shows all reports for the actual view, model and record.

E-Mail...:

Shows all email reports for the actual view, model and record.

Print...:

Shows all print actions for the actual view, model and record.

Copy URL:

Copy the URL of the form into the clipboard.

Export Data...:

Export of current/selected records into *CSV*-file or open it in Excel.

- Predefined exports
  - Choose preferences of already saved exports.
- All Fields: Fields available from the model.
- Fields to export: Defines the specific fields to export.
- Options:
  - Save: Save export as a CSV file.

  - Open: Open export in spread sheet application.
- Add field names: Add a header row with field names to the export data.
- Actions:
  - Add: Adds selected fields to *Fields to export*.
  - Remove: Removes selected fields from *Fields to export*.
  - Clear: Removes all fields from *Fields to export*.
  - Save Export: Saves field mapping to a *Predefined export* with a name.
  - Delete Export: Deletes a selected *Predefined export*.
  - OK: Exports the data (action depending on *Options*).
  - Cancel

Import Data...:

Import records from *CSV*-file.

- All Fields: Fields available in the model (required fields are marked up).
- Fields to Import: Exact sequence of all columns in the CSV file.
- File to Import: File *Dialog* for choosing a CSV file to import.
- CSV Parameters: Setup specific parameters for chosen CSV file.
  - Field Separator: Character which separates CSV fields.
  - Text Delimiter: Character which encloses text in CSV.
  - Encoding: *Character Encoding* of CSV file.
  - Lines to Skip: Count of lines to skip a headline or another offset.
- Actions:
  - Add: Adds fields to *Fields to Import*.
  - Remove: Deletes fields from *Fields to Import*.
  - Clear: Removes all fields from *Fields to Import*.
  - Auto-Detect: Tries to auto detect fields in the CSV *File to Import*.
  - OK: Proceeds the data import.
  - Cancel

Widgets

There are a several widgets used on Tryton in client side. The follow sections will explains some of them.

Date/DateTime/Time Widgets

These widgets have several key shortcuts to quickly modify the value. Each key increases if lower case or decreases if upper case:

- *y*: by one year
- *m*: by one month

- *w*: by one week
- *d*: by one day
- *h*: by one hour
- *i*: by one minute
- *s*: by one second

Search Widget

The search widget adds the ability to easily search for records on the current tab. This widget is visible only on *Tree View*.

The Syntax

A query is composed of 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 *like* or *equal* depending on the type of the field. The default operator is = except for fields of type *char*, *text* and *many2one* which is *ilike*.

Field Names

All field names shown in the *Tree View* can be searched. Field names must be followed by a *:*

For example: `Name:`

If the field name contains spaces, it is possible to escape it using double quotes.

For example: `"Receivable Today":`

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`

Note: The *ilike* operator is never explicit and % is appended to the value to make it behaves like *starts with*

Values

The format of the value depends on the type of the field. A list of values can be set using *;* as separator.

For example: `Name: Michael; Pam`

It will find all records having the *Name* starting with *Michael* or *Pam*.

A range of number values can be set using ...

For example: `Amount: 100..500`

It will find all records with *Amount* between *100* and *500* included.

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`

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)`

RichText Editor

This feature create a rich text editor with various features that allow for text formatting. The features are:

- Bold: On/off style of bold text
- Italic: On/off style of italic text
- Underline: On/off style of underline text
- Choose font family: Choice from a combo box the desired font family
- Choose font size: Choice from a combo box the desired size font
- Text justify: Choice between four options for alignment of the line (left, right, center, fill)
- Background color: Choose the background color of text from a color palette
- Foreground color: Choose the foreground color of text from a color palette

Besides these features, it can change and edit text markup. The text markup feature has a similar HTML tags and is used to describe the format specified by the user and is a way of storing this format for future opening of a correct formatted text. The tags are explain follows:

- Bold: Tag *b* is used, i.e. text
- Italic: Tag *i* is used, i.e. text
- Underline: Tag *u* is used, i.e. <u>text</u>
- Font family: It is a attribute *font-family* for *span* tag, i.e. text
- Font size: It is a attribute *size* for *span* tag, i.e. text
- Text Justify: For justification text is used paragraph tag *p*. The paragraph tag is used to create new lines and the alignment is applied across the board. Example: some text
- Background color: It is a attribute *background* for *span* tag, i.e. text
- Foreground color: It is a attribute *foreground* for *span* tag, i.e. text

CSS

The client can be styled using the file *theme.css*.

Here are the list of custom selectors:

- *.readonly*: readonly widget or label
- *.required*: widget or label of required field
- *.invalid*: widget for which the field value is not valid
- *headerbar.profile-<name>*: the name of the connection profile is set on the main window

For more information about style option see GTK+ CSS

Appendix

Configuration File

```
~/.config/tryton/x.y/tryton.conf      # General configuration
~/.config/tryton/x.y/accel.map        # Accelerators configuration
~/.config/tryton/x.y/known_hosts      # Fingerprints
~/.config/tryton/x.y/ca_certs         # Certification Authority (http://docs.python.
↪org/library/ssl.html#ssl-certificates)
~/.config/tryton/x.y/profiles.cfg     # Profile configuration
~/.config/tryton/x.y/plugins          # Local user plugins directory
~/.config.tryton/x.y/theme.css        # Custom CSS theme
```

Note: ~ means the home directory of the user. But on Windows system it is the *APPDATA* directory.

Glossary

Actions An *action* is a function which is triggered by a user intervention. *Actions* are called from activating menu items or pushing buttons. Actions often provide *Wizards*.

Board The *board* is a type of *Views* able to handle other views. This view type is not documented or not used for now.

Character Encoding See [WP-ENCOD]

CSV File format for Comma Separated Values. See [WP-CSV]

Data *Data* means information content produced by users.

Dialog A *dialog* is a *Popup* window, which overlays other windows and request user interaction. *Dialogs* are used to set up special *Actions*.

Fields *Fields* are attributes of a *data object*. *Fields* are represented as table fields in relational databases.

Form The *form* is the general type of *Views* used in Tryton. The *form* provides several modes for presenting *Data*:

- *Form View*
- *Tree View*
- *Graph View*

Form View The *form* is a mode of *Views*, which displays single *Records* of data.

Graph View *Graph view* is a mode of *Views* to show sets of data in a diagram. *Graph views* can be pie-charts or bar-charts.

Main Frame The *main frame* is a huge part arranged in the center of the *Tryton Client*. *Using the Tryton client* means mainly using the *main frame* part. It contains *Tabs* to organize and to show different *Views*.

Model A *model* describes how data is represented and accessed. Models formally define records and relationships for a certain domain of interest.

Modules *Modules* are enclosed file packages for the *Tryton Server*. A *Module* defines the *Model*, the presentation of the information (*Views*), functions, *Actions* and default presets. Additionally *modules* may provide standardized data like ISO names for countries. *Modules* in Tryton are build up generically. That is, they are constructed as simple as possible to provide the desired functionality.

Plugins A *plugin* is an add-on module for the *Tryton Client*.

Popup A small window which pops up the main window.

Records A *record* is a singular dataset in a *Model*. *Records* are represented as lines or *records* in a relational database table.

Tabs *Tabs* are *Widgets* to arrange different contents side by side. They are used to switch quickly between different domains of interest. Tryton uses *tabs* in two layer:

- A tabbed *Main Frame*.
- Tabs inside *Views*.

The main frame consists of *tabs* that embed the main menu and all views to an appropriate *Model*. The other type of *tabs* is used inside of *Views* to split them into visual domains of the same model. These *tabs* are used for structuring contents of one model to different sub headings.

Three-Tiers A *three-tiers* application framework like Tryton, is build up of three different software components:

1. The storage or data tier.
2. The logic or application tier.
3. The presentation tier.

The storage tier in the Tryton framework is provided by the PostgreSQL database engine. The application logic tier is provided by *Tryton Server* and its *Modules*. The presentation tier is mainly provided by the *Tryton*

Client. In a *three tiers* framework, the presentation tier (client) never connects directly to the storage tier. All communication is controlled by the application tier.

Tree View *Tree view* is a mode of *Views* showing sets of *Data*. *Tree views* can be flat lists or tables as well as tree-like nested lists.

Tryton Client The *Tryton Client* application is the graphical user interface (GUI) of the *Tryton Server*.

Tryton Server The *Tryton server* is the application or logic tier in the *Three-Tiers* application platform *Tryton*. The *Tryton server* connects the underlying application logic of the different *Modules* with corresponding database records. The *Tryton server* provides different interfaces to present the generated information:

- *Tryton Client*: (graphical user interface GUI)
- XMLRPC see [WP-XMLRPC]
- WebDAV see [WP-WebDAV]
- OpenOffice

Views A *view* is the visual presentation of *Data*. *Views* resides inside *Tabs* in the *Main Frame* of the *Tryton Client*. There are two general types of *views* in Tryton:

1. *Form*
2. *Board*

Each of the view types has different modes to show data. *Views* are built of several *Widgets* and provide often additional *Actions*. It is also possible to present the same data in different view modes alternately.

Widgets A *Widget* is a visual element of a graphical user interface (GUI). Some *Widgets* solely show information, others allow manipulation from user side. Example *Widgets* are buttons, check-boxes, entry-boxes, selection lists, tables, lists, trees, ...

Wizards *Wizards* define stateful sequences of interaction to proceed complex *Actions*. A *wizard* divides the complexity of some actions into several user guided steps.

References

4.2 Tryton Web Client

Tryton is a Graphical User Interface to Tryton based on GTK and Python.

In progress.

4.2.1 Contents

4.3 Tryton Server

4.3.1 First steps

- **Installation:** *Installation* | *Configuration* | *Setup a database* | *Start the server*

4.3.2 The model layer

- **Models:** *Model syntax* | *Field types* | *Domain syntax* | *Access rights* | *Triggers*

4.3.3 The view layer

- **Views:** *View types* | *Extension*
- **Reports:** *Report definition*

4.3.4 The development process

- **Modules** *Module definition*

4.3.5 Contents

Using trytond

Introduction to all the key parts of trytond:

How to install Tryton

Install Tryton

There are three easy options to install Tryton:

- Install the version provided by your operating system distribution. This is the quickest and recommended option for those who has operating system that distributes Tryton.
- Install an official release. Once you've downloaded and unpacked a trytond source release, enter the directory where the archive was unpacked, and run: `python setup.py install`

  For advanced options, please refer to the easy_install and/or the distutils documentation:

  - http://setuptools.readthedocs.io/en/latest/easy_install.html
  - http://docs.python.org/inst/inst.html
- Without installation, just run `bin/trytond` from where the archive was unpacked.

Warning: Note that you may need administrator/root privileges for this step, as this command will by default attempt to install trytond to the Python site-packages directory on your system.

Configuration file for Tryton

The configuration file controls some aspects of the behavior of Tryton. The file uses a simple ini-file format. It consists of sections, led by a *[section]* header and followed by *name* = *value* entries:

```
[database]
uri = postgresql://user:password@localhost/
path = /var/lib/trytond
```

For more information see ConfigParser.

The default value of any option can be changed using environment variables with names using this syntax: *TRYTOND_<SECTION>__<NAME>*.

Sections

This section describes the different main sections that may appear in a Tryton configuration file, the purpose of each section, its possible keys, and their possible values. Some modules could request the usage of other sections for which the guideline asks them to be named like their module.

web

Defines the behavior of the web interface.

listen

Defines the couple of host (or IP address) and port number separated by a colon to listen on.

Default *localhost:8000*

hostname

Defines the hostname to use when generating a URL when there is no request context available, for example during a cron job.

root

Defines the root path served by *GET* requests.

Default: Under the *www* directory of user's home running *trytond*.

num_proxies

The number of proxy servers in front of *trytond*.

Default: 0

cache_timeout

The cache timeout in seconds.

Default: 12h

cors

The list (one per line) of origins allowed for Cross-Origin Resource sharing.

database

Defines how the database is managed.

uri

Contains the URI to connect to the SQL database. The URI follows the RFC-3986. The typical form is:

    database://username:password@host:port/

Default: The value of the environment variable *TRYTOND_DATABASE_URI* or *sqlite://* if not set.

The available databases are:

PostgreSQL

pyscopg2 supports two type of connections:

- TCP/IP connection: *postgresql://user:password@localhost:5432/*
- Unix domain connection: *postgresql://username:password@/*

SQLite

The only possible URI is: *sqlite://*

path

The directory where Tryton stores files and so the user running *trytond* must have write access on this directory.

Default: The *db* folder under the user home directory running *trytond*.

list

A boolean value to list available databases.

Default: *True*

retry

The number of retries when a database operational error occurs during a request.

Default: *5*

language

The main language of the database that will be used for storage in the main table for translations.

Default: *en*

default_name

The name of the database to use for operations without a database name. Default: *template1* for PostgreSQL, *:memory:* for SQLite.

request

max_size

The maximum size in bytes of unauthenticated request (zero means no limit).

Default: 2MB

max_size_authenticated

The maximum size in bytes of an authenticated request (zero means no limit).

Default: 2GB

cache

Defines size of various cache.

model

The number of different model kept in the cache per transaction.

Default: *200*

record

The number of record loaded kept in the cache of the list. It can be changed locally using the *_record_cache_size* key in `Transaction.context`.

Default: *2000*

field

The number of field to load with an *eager* `Field.loading`.

Default: *100*

clean_timeout

The minimum number of seconds between two cleanings of the cache. If the value is 0, the notification between processes will be done using channels if the back-end supports them.

Default: *300*

queue

worker

Activate asynchronous processing of the tasks. Otherwise they are performed at the end of the requests.

Default: *False*

table

This section allows to override the default generated table name for a `ModelSQL`. The main goal is to bypass limitation on the name length of the database backend. For example:

```
[table]
account.invoice.line = acc_inv_line
account.invoice.tax = acc_inv_tax
```

ssl

Activates SSL on all network protocols.

Note: SSL is activated by defining privatekey. Please refer to SSL-CERT on how to use private keys and certficates.

privatekey

The path to the private key.

certificate

The path to the certificate.

email

Note: Email settings can be tested with the *trytond-admin* command

uri

The SMTP-URL to connect to the SMTP server which is extended to support SSL and STARTTLS. The available protocols are:

- *smtp*: simple SMTP
- *smtp+tls*: SMTP with STARTTLS
- *smtps*: SMTP with SSL

The uri accepts the following additional parameters:

- *local_hostname*: used as FQDN of the local host in the HELO/EHLO commands, if omited it will use the value of *socket.getfqdn()*.
- *timeout*: A number of seconds used as timeout for blocking operations. A *socket.timeout* will be raised when exceeded. If omited the default timeout will be used.

Default: *smtp://localhost:25*

from

Defines the default *From* address (using RFC-822) for emails sent by Tryton.

For example:

```
from: "Company Inc" <info@example.com>
```

session

authentications

A comma separated list of login methods to use to authenticate the user. By default, Tryton supports only the *password* method which compare the password entered by the user against a stored hash. But other modules can define new methods (please refers to their documentation). The methods are tested following the order of the list.

Default: *password*

max_age

The time in seconds that a session stay valid.

Default: *2592000* (30 days)

timeout

The time in seconds without activity before the session is no more fresh.

Default: *300* (5 minutes)

max_attempt

The maximum authentication attempt before the server answers unconditionally *Too Many Requests* for any other attempts. The counting is done on all attempts over a period of *timeout*.

Default: *5*

max_attempt_ip_network

The maximum authentication attempt from the same network before the server answers unconditionally *Too Many Requests* for any other attempts. The counting is done on all attempts over a period of *timeout*.

Default: *300*

ip_network_4

The network prefix to apply on IPv4 address for counting the authentication attempts.

Default: *32*

ip_network_6

The network prefix to apply on IPv6 address for counting the authentication attempts.

Default: *56*

password

length

The minimal length required for the user password.

Default: *8*

forbidden

The path to a file containing one forbidden password per line.

entropy

The ratio of non repeated characters for the user password.

Default: *0.75*

reset_timeout

The time in seconds until the reset password expires.

Default: *86400* (24h)

passlib

The path to the INI file to load as CryptContext. If not path is set, Tryton will use the schemes *bcrypt* or *pbkdf2_sha512*.

Default: *None*

attachment

Defines how to store the attachments

filestore

A boolean value to store attachment in the *FileStore*.

Default: *True*

store_prefix

The prefix to use with the *FileStore*.

Default: *None*

bus

allow_subscribe

A boolean value to allow clients to subscribe to bus channels.

Default: *False*

url_host

If set redirects bus requests to the host URL.

long_polling_timeout

The time in seconds to keep the connection to the client opened when using long polling for bus messages

Default: *300*

cache_timeout

The number of seconds a message should be kept by the queue before being discarded.

Default: *300*

select_timeout

The timeout duration of the select call when listening on a channel.

Default: *5*

html

src

The URL pointing to TinyMCE editor.

Default: *https://cloud.tinymce.com/stable/tinymce.min.js*

plugins

The space separated list of TinyMCE plugins to load. It can be overridden for specific models and fields using the names: *plugins-<model>-<field>* or *plugins-<model>*.

Default: ‘‘

css

The JSON list of CSS files to load. It can be overridden for specific models and fields using the names: *css-<model>-<field>* or *css-<model>*.

Default: *[]*

class

The class to add on the body. It can be overridden for specific models and fields using the names: *class-<model>-<field>* or *class-<model>*.

Default: ‘’

wsgi middleware

The section lists the WSGI middleware class to load. Each middleware can be configured with a section named *wsgi <middleware>* containing *args* and *kwargs* options.

Example:

```
[wsgi middleware]
ie = werkzeug.contrib.fixers.InternetExplorerFix

[wsgi ie]
kwargs={'fix_attach': False}
```

How to setup a database

The database section of the *configuration* must be set before starting.

Create a database

Depending of the database backend choosen, you must create a database (see the documentation of the choosen backend). The user running *trytond* must be granted the priviledge to create tables. For backend that has the option, the encoding of the database must be set to *UTF-8*.

Initialize a database

A database can be initialized using this command line:

```
trytond-admin -c <config file> -d <database name> --all
```

At the end of the process, *trytond-admin* will ask to set the password for the *admin* user.

Update a database

To upgrade to a new series, the command line is:

```
trytond-admin -c <config file> -d <database name> --all
```

Warning: Prior to upgrade see if there is no manual action to take on the migration topic.

To activate a new language on an existing database, the command line is:

```
trytond-admin -c <config file> -d <database name> --all -l <language code>
```

Once activated, the language appears in the user preferences.

When installing new modules, the list of modules must be updated with:

```
trytond-admin -c <config file> -d <database name> --update-modules-list
```

Once updated, the new modules can be activated from the client or activated with:

```
trytond-admin -c <config file> -d <database name> -u <module name> --activate-
↪dependencies
```

Logging configuration

Without any configuration, trytond writes ERROR messages to standard output. For each verbose flag set, the log level decreases.

Logs can be configured using a configparser-format file. The filename can be specified using trytond `logconf` parameter.

Example

This example allows to write INFO messages on standard output and on a disk log file rotated every day.

```
[formatters]
keys=simple

[handlers]
keys=rotate,console

[loggers]
keys=root

[formatter_simple]
format=%(asctime)s] %(levelname)s:%(name)s:%(message)s
datefmt=%a %b %d %H:%M:%S %Y

[handler_rotate]
class=handlers.TimedRotatingFileHandler
args=('/tmp/tryton.log', 'D', 1, 30)
formatter=simple

[handler_console]
class=StreamHandler
formatter=simple
args=(sys.stdout,)

[logger_root]
level=INFO
handlers=rotate,console
```

How to start the server

Web service

You can start the default web server bundled in Tryton with this command line:

```
trytond -c <config file>
```

The server will wait for client connections on the interface defined in the *web* section of the *configuration*.

Note: When using multiple config files the order is importart as last entered files will override the items of first files

WSGI server

If you prefer to run Tryton inside your own WSGI server instead of the simple server of Werkzeug, you can use the application *trytond.application.app*. Following environment variables can be set:

- *TRYTOND_CONFIG*: Point to *configuration* file.
- *TRYTOND_LOGGING_CONFIG*: Point to *logging* file.
- *TRYTOND_LOGGING_LEVEL*: An integer to set the default logging level (default: *ERROR*).
- *TRYTOND_COROUTINE*: Use coroutine for concurrency.
- *TRYTOND_DATABASE_NAMES*: A list of database names in CSV format, using python default dialect.

Warning: You must manage to serve the static files from the web root.

Coroutine server

The Werkzeug server uses thread for concurrency. This is not optimal for the long-polling request on the *bus* as each client consumes permanently one thread. You can start the server with coroutine using the option *–coroutine*.

Cron service

If you want to run some *scheduled actions*, you must also run the cron server with this command line:

```
trytond-cron -c <config file> -d <database>
```

The server will wake up every minutes and preform the scheduled actions defined in the *database*.

Worker service

If you want to use a pool of workers to run *asynchronously some tasks*, you must activate the worker in the *queue* section of the *configuration* and run the worker manager with this command line:

```
trytond-worker -c <config file> -d <database>
```

The manager will dispatch tasks from the queue to a pool of worker processes.

Services options

You will find more options for those services by using *–help* arguments.

Models

A model represents a single business logic or concept. It contains fields and defines the behaviors of the record. Most of the time, each model stores records in a single database table.

The basics:

- Each model is a Python class that subclasses one of `trytond.model.model.Model`.
- *Fields* are defined as model attributes.
- Tryton generates the table definitions
- Tryton provides an API following the active record pattern to access the records.

Example

This example defines a `Party` model which has a `name` and a `code` fields:

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

class Party(ModelSQL, ModelView):
    "Party"
    __name__ = "party.party"
    name = fields.Char('Name')
    code = fields.Char('Code')
```

The class must be registered in the *Pool* by the *register()* method of the *module*. Model classes are essentially data mappers to records and Model instances are records.

Model attributes define meta-information of the model. They are class attributes starting with an underscore. Some model properties are instance attributes allowing to update them at other places in the framework.

Default value of fields

When a record is created, each field, which doesn't have a value specified, is set with the default value if exists.

The following class method:

```
Model.default_<field name>()
```

Return the default value for `field name`.

This example defines an `Item` model which has a default `since`:

```
import datetime

from trytond.model import ModelView, ModelSQL, fields

class Item(ModelSQL, ModelView):
    "Item"
    __name__ = 'item'
    since = fields.Date('since')

    @classmethod
    def default_since(cls):
        return datetime.date.today()
```

See also method `Model.default_get`: *default_get*

on_change of fields

Tryton allows developers to define methods that can be called once a field's value has changed by the user this is the *on_change* method. The method has the following name:

```
Model.on_change_<field name>
```

This is an instance method, an instance of `Model` will be created by using the values from the form's fields specified by the `on_change` list defined on the field. Any change made on the instance will be pushed back to the client-side record.

There is also a way to define a method that must update a field whenever any field from a predefined list is modified. This list is defined by the *on_change_with* attribute of the field. The method that will be called has the following name:

```
Model.on_change_with_<field_name>
```

Just like for the classic `on_change`, an instance of `Model` will be created by using the values entered in the form's fields specified by the `on_change_with` attribute. The method must return the new value of the field to push back to the client-side record.

Backend Types supported

This table give a comprehensive list of the SQL Types that are expected to be supported by the database backends. If the type is not supported then the backend will have to emulate the behavior described here.

The columns are in the following order:

- The SQL type[1] representing the field
- The python type expected on input
- The python type received on output

None will represent the *NULL* value and vice versa, it can be used as input or output for any SQL type.

SQL Type	Python input type	Python output type
BOOL	bool	bool
INTEGER	int	int
BIGINT	int / long[2]	int / long[2]
FLOAT	float / int / long[2]	float
NUMERIC	decimal.Decimal	decimal.Decimal
VARCHAR / *VARCHAR(length)*	str / unicode[3]	str / unicode[3]
TEXT	str / unicode[3]	str / unicode[3]
TIMESTAMP	datetime.datetime	datetime.datetime
DATETIME	datetime.datetime without microseconds[4]	datetime.datetime without microseconds[4]
DATE	datetime.date	datetime.date
TIME	datetime.time	datetime.time
INTERVAL	datetime.timedelta	datetime.timedelta
BLOB	bytes	bytes

Domain

Domains represent a set of records. A domain is a list of none or more clauses. A clause is a condition, which returns true or false. A record belongs to a domain, when the final result of the list of clauses returns true.

Syntax

The definition of a simple domain with one clause is represented by this pattern:

[1] Corresponding to the SQL 92 standard or to a PostgreSQL type.
[2] in python 2 integers over *sys.maxint* are represented by the *long* type
[3] str when using python 3 ; unicode when using python 2
[4] Datetime objects are not localized to any timezone

```
domain = [(<field name>, <operator>, <operand>)]
```

<field name> Is the name of a *fields* or a *pyson* statement, that evaluates to a string.

A field of type *Many2One* or *Many2Many* or *One2Many* or *One2One* or *Reference* can be dereferenced to related models. This is illustrated by the following example:

```
domain = [('country.name', '=', 'Japan')]
```

The number of *dots* in a clause is not limited.

Warning: For *trytond.model.fields.Reference*, an extra ending clause is needed to define the target model to join, for example:

```
domain = [('origin.party.name', '=', 'John Doe', 'sale.sale')]
```

A field of type *Dict* can be searched by key also by using one *dot*. For example:

```
domain = [('attributes.color', '=', 'yellow')]
```

Warning: Order comparison of *date* and *datetime* types is not supported.

operator Is an operator out of *Domain Operators* or a *pyson* statement, that evaluates to a domain operator string.

operand Is an operand or a *pyson* statement. The type of operand depends on the kind of <field name>.

The definition of an empty domain is:

```
domain = []
```

An empty domain without clauses will always return all *active* records. A record is active, when its appropriate *Model* contains a *Boolean* field with name `active`, and set to true. When the appropriate *Model* does not contain a *Boolean* field with name `active` all records are returned.

A domain can be setup as a combination of clauses, like shown in this pattern:

```
domain = [
    ('field name1', 'operator1', 'operand1'),
    ('field name2', 'operator2', 'operand2'),
    ('field name3', 'operator3', 'operand3'),]
```

The single clauses are implicitly combined with a logical AND operation.

In the domain syntax it is possible to provide explicitly the combination operation of the clauses. These operations can be AND or OR. This is illustrated by the following pattern:

```
domain = [ 'OR', [
            ('field name1', 'operator1', 'operand1'),
            ('field name2', 'operator2', 'operand2'),
        ], [
            ('field name3', 'operator3', 'operand3'),
        ],]
```

Here the domain is evaluated like this: `((clause1 AND clause2) OR clause3)`. Please note that the `AND` operation is implicit assumed when no operator is given. While the `OR` operation must be given explicitly. The former pattern is equivalent to the following completely explicit domain definition:

```
domain = [ 'OR',
              [ 'AND', [
                        ('field name1', 'operator1', 'operand1'),
                    ], [
                        ('field name2', 'operator2', 'operand2'),
                    ],
              ], [
                    ('field name3', 'operator3', 'operand3'),
          ],]
```

Obviously the use of the implicit `AND` operation makes the code more readable.

Domain Operators

The following operators are allowed in the domain syntax. `<field name>`, `<operator>` and `<operand>` are dereferenced to their values. The description of each operator follows this pattern, unless otherwise noted:

```
(<field name>, <operator>, <operand>)
```

=

Is a parity operator. Returns true when `<field name>` equals to `<operand>`.

!=

Is an imparity operator. It is the negation of the = operator.

like

Is a pattern matching operator. Returns true when `<field name>` is contained in the pattern represented by `<operand>`.

In `<operand>` an underscore (_) matches any single character, a percent sign (`%`) matches any string with zero or more characters. To use _ or `%` as literal, use the backslash \ to escape them. All matching is case sensitive.

not like

Is a pattern matching operator. It is the negation of the *like* operator.

ilike

Is a pattern matching operator. The same use as *like* operator, but matching is case insensitive.

not ilike

Is a pattern matching operator. The negation of the *ilike* operator.

in

Is a list member operator. Returns true when `<field name>` is in `<operand>` list.

not in

Is a list non-member operator. The negation of the *in* operator.

<

Is a *less than* operator. Returns true for type string of `<field name>` when `<field name>` is alphabetically sorted before `<operand>`.

Returns true for type number of `<field name>` when `<field name>` is less than `<operand>`.

>

Is a *greater than* operator. Returns true for type string of `<field name>` when `<field name>` is alphabetically sorted after `<operand>`.

Returns true for type number of `<field name>` when `<field name>` is greater `<operand>`.

<=

Is a *less than or equal* operator. Returns the same as using the < operator, but also returns true when `<field name>` is equal to `<operand>`.

>=

Is a *greater than or equal* operator. Returns the same as using the > operator, but also returns true when `<field name>` is equal to `<operand>`.

child_of

Is a parent child comparison operator. Returns true for records that are a child of `<operand>`. `<operand>` is a list of `ids` and `<field name>` must be a `many2one` or a `many2many`. In case `<field name>` is not linked to itself, the clause pattern extends to:

```
(<field name>, ['child_of'|'not_child_of'], <operand>, <parent field>)
```

Where `<parent field>` is the name of the field constituting the `many2one` on the target model.

not child_of

Is a parent child comparison operator. It is the negation of the *child_of* operator.

parent_of

Is a parent child comparison operator. It is the same as *child_of* operator but if `<field name>` is a parent of `<operand>`.

not parent_of

Is a parent child comparison operator. It is the negation of this *parent_of* operator.

where

Is a *trytond.model.fields.One2Many* / *trytond.model.fields.Many2Many* domain operator. It returns true for every row of the target model that match the domain specified as `<operand>`.

not where

Is a *trytond.model.fields.One2Many* / *trytond.model.fields.Many2Many* domain operator. It returns true for every row of the target model that does not match the domain specified as `<operand>`.

PYSON

PYSON is the PYthon Statement and Object Notation. It is a lightweight domain specific language for the general representation of statements. PYSON is used to encode statements which can be evaluated in different programming languages, serving for the communication between trytond and any third party software. A PYSON parser can easily be implemented in other programming languages. So third party softwares do not need to depend on Python to be able to fully communicate with the Tryton server.

PYSON is a deterministic algorithm which will always succeed to evaluate statements. There is a default behavior for unknown values. It is statically typed and checked on instantiation.

There is also a *reference documentation of the API*.

Syntax

The syntax of a PYSON statement follows this pattern:

```
Statement(argument1[, argument2[, ...]])
```

where arguments can be another statement or a value. The evaluation direction is inside out, deepest first.

PYSON Examples

Given the PYSON statement:

```
Eval('active_id', -1)
```

`Eval()` checks the evaluation context for the variable `active_id` and returns its value or `-1` if not defined. A similar expression in Python looks like this:

```
'active_id' in locals() and active_id or -1
```

Given the PYSON statement:

```
Not(Bool(Eval('active')))
```

`Eval()` checks the evaluation context for a variable `active` and returns its value to `Bool()` or `''` if not defined. `Bool()` returns the corresponding boolean value of the former result to `Not()`. `Not()` returns the boolean negation of the previous result. A similar expression in Python looks like this:

```
'active' in locals() and active == False
```

Given the PYSON statement:

```
Or(Not(Equal(Eval('state'), 'draft')), Bool(Eval('lines')))
```

In this example are the results of two partial expressions `Not(Equal(Eval('state'), 'draft'))` and `Bool(Eval('lines'))` evaluated by a logical *OR* operator. The first expression part is evaluated as follow: When the value of `Eval('state')` is equal to the string `'draft'` then return true, else false. `Not()` negates the former result. A similar expression in Python looks like this:

```
'states' in locals() and 'lines' in locals() \
        and state != 'draft' or bool(lines)
```

Given the PYSON statement:

```
If(In('company', Eval('context', {})), '=', '!=')
```

In this example the result is determined by an if-then-else condition. `In('company', Eval('context', {}))` is evaluated like this: When the key `'company'` is in the dictionary `context`, returns true, otherwise false. `If()` evaluates the former result and returns the string `'='` if the result is true, otherwise returns the string `'!='`. A similar expression in Python looks like this:

```
'context' in locals() and isinstance(context, dict) \
        and 'company' in context and '=' or '!='
```

Given the PYSON statement:

```
Get(Eval('context', {}), 'company', 0))
```

`Eval()` checks the evaluation context for a variable `context` if defined, return the variable `context`, otherwise return an empty dictionary `{}`. `Get()` checks the former resulting dictionary and returns the value of the key `'company'`, otherwise it returns the number `0`. A similar expression in Python looks like this:

```
'context' in locals() and context.get('company', 0) or 0
```

Access Rights

There are 5 levels of access rights: *Model* , *Actions*, *Field*, *Button* and *Record Rule*. They are based on the user's group membership. If any of those levels are violated, an error is raised.

The access rights are checked if the *`Transaction.context`* has the key `_check_access` set to `True` (set by default by *`RPC.check_access`*) and if the *`user`* is not *root*.

Warning: The record rules are always enforced regardless of the *_check_access* value.

Model

They are defined by records of `ir.model.access` which define for each couple of model and group, the `read`, `write`, `create` and `delete` permission. The permissions are related to the *`ModelStorage`* methods with the same name and on *`search()`* using the `read` permission.

If any group the user belongs to has the checked permission activated, then the user is granted this permission.

If there is no record for the model, then access is granted to all users.

Note: Relation fields for which the user has no read access are automatically removed from the *views*.

Actions

Each `ir.action` has a `groups` field which contains a list of user groups that are allowed to see and launch it.

There is a special case for *wizard* for which the read access on the model is also checked and also the write access if there is no groups linked.

Field

They are defined by records of `ir.model.field.access` and work like those for *Model* but are applied to *fields*.

Note: Fields for which the user has no read access are automatically removed from the *views*.

Button

For each button of a model the records of *ir.model.button* define the list of groups that are allowed to call it. The user only needs to belong to one of the groups to be granted the permission to use it.

If no group is defined for a button, the `write` permission to the model is checked instead.

The `read` permission to the model is always enforced.

Note: Buttons for which the user has no access are marked readonly.

Button Rule

The *ir.model.button* can contain a list of rules which define how many different users must click on the button. Each rule, for which the condition is met, must be passed to actually trigger the action. The counter can be reset when another defined button is clicked.

Record Rule

The record rules are conditions that records must meet for the user to be granted permission to use them. They are defined by records of *ir.rule.group* which contains:

- a model on which it applies
- the permissions granted
- a set of user groups to which the rule applies
- a global flag to always enforce
- a default flag to add to all users
- a list of *ir.rule* with a *domain* to select the records to which the rule applies.

A rule group matches a record if the record is validated by at least one of the domains. The access is granted to a record:

- if the user belongs to a group which has at least one matching rule group that has the permission,
- or if there is a default matching rule group with the permission,
- or if there is a global matching rule group with the permission.

Otherwise the access is denied if there is any matching rule group.

Note: Records for which the user has no `read` access are filtered out from the *search()* result.

User Errors and Warnings

When processing a request, you can stop the flow by raising an exception that will be displayed to the user as an error message or a warning. They are respectively *UserError* and *UserWarning*.

User Errors

An error displays a message and optionally a description to the user.

Example:

```
from trytond.exceptions import UserError
from trytond.model import Model

class MyModel(Model):
    "My Model"
    __name__ = 'my_model'

    def process(self):
        if check_failed:
            raise UserError("You cannot process.", "becauseâ€¦")
```

Note: They are often used in combination with *gettext()* to translate the messages.

User Warnings

A warning displays a confirmation message with optionally a description to the user. The user can decide to continue so the request is processed again without stopping at the warning. Otherwise the user can cancel its request. The warning instance is identified by a name which allows to skip it the next time it is checked, that's why they often contain data like the id of a record.

Example:

```
from trytond.exceptions import UserWarning
from trytond.model import Model
from trytond.pool import Pool

class MyModel(Model):
    "My Model"
    __name__ = 'my_model'

    def process(self):
        Warning = Pool().get('res.user.warning')
        warning_name = 'mywarning,%s' % self
        if Warning.check(warning_name):
            raise UserWarning(warning_name, "Process cannot be canceled.")
```

Triggers

Triggers allow to define methods of `trytond.model.model.Model` that are called at the end of the transaction when one of those events happen to a record:

- On Creation
- On Modification
- On Deletions
- On Time: When a condition changes over time.

The method signature is:

```
<method name>(cls, records, trigger)
```

Where *records* is the list of records that triggered the event and *trigger* is the *ir.trigger* instance which is triggered.

Triggers are defined by records of *ir.trigger*. Each record must define a pyson condition which will be evaluated when the event occurs. Only those records for which the condition is evaluated to true will be processed by the trigger with the exception of modification triggers which will only process the records for which the condition is evaluated to false before and evaluated to true after the modification.

Actions

Actions are used to describe specific behaviors in the client.

There are four types of actions:

- Report
- Window
- Wizard

- URL

Keyword

Keywords define where to display the action in the client.

There are five places:

- Open tree (*tree_open*)
- Print form (*form_print*)
- Action form (*form_action*)
- Form relate (*form_relate*)
- Open Graph (*graph_open*)

Report

Window

The window action describe how to create a new tab in the client.

View

Domain

The window action could have a list of domains which could be activated on the view. The boolean field count indicates if the client must display the number of records for this domain.

> **Warning:** The counting option must be activated only on domains which have not too much records otherwise it may overload the database.

Wizard

URL

Views

The views are used to display records of an `ModelView` to the user.

In Tryton, `ModelView` can have several views. An *action* opens a window and defines which view to show.

The views are built from XML that is stored in the *view* directory of the module or in the databases thanks to the model ir.ui.view.

So generally, they are defined in xml files with this kind of xml where name is the name of the XML file in the *view* directory:

```
<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"/-->
    <!--field name="field_childs">field name</field-->
    <field name="name">view_name</field>
</record>
```

There are different types of views:

- Form
- Tree
- List-Forms
- Graph
- Board
- Calendar

Form view

The RNG describing the xml of a form view is stored in trytond/ir/ui/form.rng. There is also a RNC in trytond/ir/ui/form.rnc.

A form view is used to display one record of an object.

Elements of the view are put on the screen following the rules:

- Elements are placed on the screen from left to right, from top to bottom, according to the order of the xml.
- The screen composed of a table with a fixed number of columns and enough rows to handle all elements.
- Elements take one or more columns when they are put in the table. If there are not enough free columns on the current row, the elements are put at the beginning of the next row.

XML description

List of attributes shared by many form elements:

- `id`: A unique identifier for the tag if there is no name attribute.
- `yexpand`: A boolean to specify if the label should expand to take up any extra vertical space.
- `yfill`: A boolean to specify if the label should fill the vertical space allocated to it in the table cell.
- `yalign`: The vertical alignment, from 0.0 to 1.0.
- `xexpand`: The same as yexpand but for horizontal space.
- `xfill`: The same as yfill but for horizontal space.
- `xalign`: The horizontal alignment, from 0.0 to 1.0.
- `colspan`: The number of columns the widget must take in the table.
- `col`: The number of columns the container must have.

  A negative value (or zero) will remove the constraint on the number of columns.

  The default value is 4.

- `states`: A string of *PYSON statement* that will be evaluated with the values of the current record.

  It must return a dictionary where keys can be:

  - `invisible`: If true, the widget will be hidden.
  - `required`: If true, the field will be required.
  - `readonly`: If true, the field will be readonly.
  - `icon`: Only for button, it must return the icon name to use or False.
  - `pre_validate`: Only for button, it contains a domain to apply on the record before calling the button.
  - `depends`: Only for button, it must return the list of field on which the button depends.
- `help`: The string that will be displayed when the cursor hovers over the widget.
- `pre_validate`: A boolean only for fields *`trytond.model.fields.One2Many`* to specify if the client must pre-validate the records using *`trytond.model.Model.pre_validate()`*.
- `completion`: A boolean only for fields *`trytond.model.fields.Many2One`*, *`trytond.model.fields.Many2Many`* and *`trytond.model.fields.One2Many`* to specifiy if the client must auto-complete the field. The default value is True.
- `factor`: A factor to apply on fields *`trytond.model.fields.Integer`*, *`trytond.model.fields.Float`* and *`trytond.model.fields.Numeric`* to display on the widget. The default value is 1.

form

Each form view must start with this tag.

- `on_write`: The name of a method on the Model of the view that will be called when a record is saved. The method must return a list of record ids that the client must reload if they are already loaded. The function must have this syntax:

  ```
  on_write(self, ids)
  ```

  Note: The method must be registered in *`trytond.model.Model.__rpc__`*.

- `col`: see in *common-attributes-col*.
- `cursor`: The name of the field that must have the cursor by default.

label

Display static string.

- `string`: The string that will be displayed in the label.
- `name`: The name of the field whose description will be used for string. Except if `string` is set, it will use this value and the value of the field if `string` is empty.
- `id`: see *common-attributes-id*.
- `yexpand`: see in *common-attributes-yexpand*.
- `yfill`: see in *common-attributes-yfill*.

- `yalign`: see in *common-attributes-yalign*.
- `xexpand`: see in *common-attributes-xexpand*.
- `xfill`: see in *common-attributes-xfill*.
- `xalign`: see in *common-attributes-xalign*.
- `colspan`: see in *common-attributes-colspan*.
- `states`: see in *common-attributes-states*.
- `help`: see in *common-attributes-help*.
- Requires that either `id` or `name` is defined.

field

Display a field of the object with the value of the current record.

- `name`: The name of the field.
- `string`: The string that will be displayed for the widget.
- `widget`: The widget that must be used instead of the default one.
- `help`: The string that will be displayed when the cursor stays over the widget.
- `width`: The minimum width the widget should request, or -1 to unset.
- `height`: The minimum height the widget should request, or -1 to unset.
- `readonly`: Boolean to set the field readonly.
- `mode`: Only for One2Many fields: it is a comma separated list, that specifies the order of the view used to display the relation. (Example: `tree,form`)
- `view_ids`: A comma separated list that specifies the view ids used to display the relation. For Many2One and Many2Many, the order should always be tree then form.
- `product`: Only for One2Many fields, a comma separated list of target field name used to create records from the cartesian product.
- `completion`: Only for Many2One fields, it is a boolean to set the completion of the field.
- `invisible`: The field will not be displayed, but it will fill cells in the table.
- `filename_visible`: Only for Binary fields, boolean that enables the display of the filename.
- `toolbar`: Only for Rich Text widget, boolean that enables the display of the Rich Text toolbar. The default value is 1.
- `yexpand`: see in *common-attributes-yexpand*.
- `yfill`: see in *common-attributes-yfill*.
- `xexpand`: see in *common-attributes-xexpand*.
- `xfill`: see in *common-attributes-xfill*.
- `colspan`: see in *common-attributes-colspan*.
- `help`: see in *common-attributes-help*.
- `pre_validate`: see in *common-attributes-pre_validate*.
- `completion`: see in *common-attributes-completion*.

- `factor`: see in *common-attributes-factor*.
- `spell`: Only for Text widgets, a *PYSON statement* that will be evaluated to the language code for which spell checking must be done.

image

Display an image.

- `name`: the image name or the field name which contains the image name. The image name must be the name of a record of *ir.ui.icon*.
- `yexpand`: see in *common-attributes-yexpand*.
- `yfill`: see in *common-attributes-yfill*.
- `colspan`: see in *common-attributes-colspan*.
- `size`: the size of the image in pixels. The default value is 48.
- `states`: see in *common-attributes-states*.
- `help`: see in *common-attributes-help*.

separator

Display a horizontal separator.

- `string`: The string that will be displayed above the separator.
- `name`: The name of the field from which the description will be used for string.
- `id`: see in *common-attributes-id*.
- `yexpand`: see in *common-attributes-yexpand*.
- `yfill`: see in *common-attributes-yfill*.
- `colspan`: see in *common-attributes-colspan*.
- `states`: see in *common-attributes-states*.
- `help`: see in *common-attributes-help*.
- Requires that either `id` or `name` is defined.

newline

Force to use a new row.

button

Display a button.

- `name`: The name of the function that will be called. The function must have this syntax:

  ```
  button(cls, records)
  ```

  The function may return an *ir.action* id or one of those client side action keywords:

- `new`: to create a new record
- `delete`: to delete the selected records
- `remove`: to remove the record if it has a parent
- `copy`: to copy the selected records
- `next`: to go to the next record
- `previous`: to go to the previous record
- `close`: to close the current tab
- `switch <view type> [<view id>]`: to switch the view
- `reload`: to reload the current tab
- `reload context`: to reload user context
- `reload menu`: to reload menu
- `icon`
- `confirm`: A text that will be displayed in a confirmation popup when the button is clicked.
- `colspan`: see in *common-attributes-colspan*.
- `states`: see in *common-attributes-states*.
- `help`: see in *common-attributes-help*.
- `keyword`: specify where will the button be displayed in the client toolbar. The valid values are the keywords starting with *form_* from *Actions* without the *form_* part.

Warning: The button should be registered on `ir.model.button` where the default value of the `string`, `confirm` and `help` attributes can be can be defined.

link

Display an *ir.action.act_window* as a button with a counter or one counter per tab. When clicked it opens the window.

- `name`: The XML id of *ir.action.act_window*.
- `colspan`: see in *common-attributes-colspan*.
- `states`: see in *common-attributes-states*.
- `icon`: The name of the icon to display.
- `empty`: If set to *hide* the button is not displayed if the counter is zero. The default is `show`.

notebook

It adds a notebook widget which can contain page tags.

- `colspan`: see in *common-attributes-colspan*.
- `states`: see in *common-attributes-states*.

page

Define a new tab inside a notebook.

- `string`: The string that will be displayed in the tab.
- `angle`: The angle in degrees between the baseline of the label and the horizontal, measured counterclockwise.
- `col`: see in *common-attributes-col*.
- `id`: see in *common-attributes-id*.
- `states`: see in *common-attributes-states*.
- Requires that either `id` or `name` is defined.

group

Create a sub-table in a cell.

- `string`: If set a frame will be drawn around the field with a label containing the string. Otherwise, the frame will be invisible.
- `rowspan`: The number of rows the group spans in the table.
- `col`: see in *common-attributes-col*.
- `expandable`: If this attribute is present the content of the group will be expandable by the user to reveal its content. A value of "1" means that the group will start expanded, a value of "0" means that the group will start unexpanded. There is no default value.
- `homogeneous`: If True all the tables cells are the same size.
- `id`: see in *common-attributes-id*.
- `yexpand`: see in *common-attributes-yexpand*.
- `yfill`: see in *common-attributes-yfill*.
- `yalign`: see in *common-attributes-yalign*.
- `xexpand`: see in *common-attributes-xexpand*.
- `xfill`: see in *common-attributes-xfill*.
- `xalign`: see in *common-attributes-xalign*.
- `colspan`: see in *common-attributes-colspan*.
- `states`: see in *common-attributes-states*.
- Requires that either `id` or `name` is defined.

hpaned, vpaned

- `position`: The pixel position of divider, a negative value means that the position is unset.
- `id`: see in *common-attributes-id*.
- `colspan`: see in *common-attributes-colspan*. The default for panes is 4 columns.

child

Contains the childs of a hpaned or vpaned.

Example

```
<form col="6">
    <label name="name"/>
    <field name="name" xexpand="1"/>
    <label name="code"/>
    <field name="code"/>
    <label name="active"/>
    <field name="active" xexpand="0" width="100"/>
    <notebook colspan="6">
        <page string="General">
            <field name="addresses" mode="form,tree" colspan="4"
                view_ids="party.address_view_form,party.address_view_tree_sequence"/>
            <label name="type"/>
            <field name="type" widget="selection"/>
            <label name="lang"/>
            <field name="lang" widget="selection"/>
            <label name="website"/>
            <field name="website" widget="url"/>
            <separator string="Categories" colspan="4"/>
            <field name="categories" colspan="4"/>
        </page>
        <page string="Accounting">
            <label name="vat_country"/>
            <field name="vat_country"/>
            <label name="vat_number"/>
            <field name="vat_number"/>
        </page>
    </notebook>
</form>
```

Tree view

The RNG that describes the xml for a tree view is stored in trytond/ir/ui/tree.rng. There is also a RNC in trytond/ir/ui/tree.rnc.

Tree view is used to display records inside a list or a tree.

It is a tree if there is a *field_childs* defined and this tree will have drag and drop activated if the *field_childs* and the *parent field* are defined in the view.

The columns of the view are put on the screen from left to right.

XML description

tree

Each tree view must start with this tag.

- `on_write`: see *form-attributes-on_write*.

- `editable`: A boolean to specify if the list is editable.
- `sequence`: The name of the field that is used for sorting. This field must be an integer and it will be updated to match the new sort order when the user uses `Drag and Drop` on list rows.
- `keyword_open`: A boolean to specify if the client should look for a tree_open action on double click instead of switching view.
- `tree_state`: A boolean to specify if the client should save the state of the tree.
- `visual`: A *PYSON statement* that will be evaluated as string `muted`, `success`, `warning` or `danger` with the context of the record to provide a visual context to the row.

field

- `name`: The name of the field.
- `readonly`: Boolean to set the field readonly.
- `widget`: The widget that must be used instead of the default one.
- `tree_invisible`: A string of *PYSON statement* that will be evaluated as boolean with the context of the view to display or not the column.
- `visual`: A *PYSON statement* that will be evaluated as string `muted`, `success`, `warning` or `danger` with the context of the record to provide a visual context to the field.
- `icon`: The name of the field that contains the name of the icon to display in the column.
- `sum`: A text for the sum widget that will be added on the bottom of list with the sum of all the fields in the column.
- `width`: Set the width of the column.
- `expand`: An integer that specifies if the column should be expanded to take available extra space in the view. This space is shared proportionaly among all columns that have their "expand" property set. Resize doesn't work if this option is enabled.
- `pre_validate`: see in *common-attributes-pre_validate*.
- `completion`: see in *common-attributes-completion*.
- `factor`: see in *common-attributes-factor*.

prefix or suffix

A `field` could contain one or many `prefix` or `suffix` that will be diplayed in the same column.

- `string`: The text that will be displayed.
- `name`: The name of the field whose value will be displayed.
- `icon`: The name of the field that contains the name of the icon to display or the name of the icon.

button

Same as in *form-button*.

Example

```
<tree sequence="sequence">
    <field name="name"/>
    <field name="percentage">
        <suffix name="percentage" string="%"/>
    </field>
    <field name="group"/>
    <field name="type"/>
    <field name="active"/>
    <field name="sequence" tree_invisible="1"/>
</tree>
```

button

Display a button.

- `string`: The string that will be displayed inside the button.
- `name`: The name of the function that will be called. The function must have this syntax:

  ```
  button(cls, records)
  ```

- `confirm`: A text that will be displayed in a confirmation popup when the button is clicked.
- `help`: see in *common-attributes-help*

List-Form view

The list-form views use the same schema as the form views.

List-forms display records as a list of editable forms.

Note: The performance of the list-form does not allow to scale well for large number of records

Graph view

The RNG that describes the xml for a graph view is stored in trytond/ir/ui/graph.rng. There is also a RNC in trytond/ir/ui/graph.rnc.

XML description

graph

Each graph view must start with this tag.

- `type`: `vbar`, `hbar`, `line`, `pie`
- `background`: an hexaecimal value for the color of the background.
- `color`: the main color.
- `legend`: a boolean to specify if the legend must be displayed.

x, y

Describe the field that must be used for axis. `x` must contain only one tag `field` and `y` must at least one but may contain many.

field

- `name`: the name of the field on the object to use.
- `string`: allow to override the string that comes from the object.
- `key`: can be used to distinguish fields with the same name but with different domain.
- `domain`: a PySON string which is evaluated with the object value as context. If the result is true the field value is added to the graph.
- `fill`: defined if the graph shall be filled.
- `empty`: defined if the line graph must put a point for missing dates.
- `color`: the color of the field.
- `interpolation`: defined how the line graph must interpolate points. The default is `linear`.
  - `constant-center`: use the value of the nearest point, see Nearest-neighbor interpolation
  - `constant-left`: use the value of the nearest left point.
  - `constant-right`: use the value of the nearest right point.
  - `linear`: see linear interpolation

Example

```
<graph string="Invoice by date" type="vbar">
  <x>
      <field name="invoice_date"/>
  </x>
  <y>
      <field name="total_amount"/>
  </y>
</graph>
```

Board view

The RNG that describes the xml for a board view is stored in trytond/ir/ui/board.rng. There is also a RNC in trytond/ir/ui/graph.rnc.

Board view is used to display multiple views at once.

Elements are put on the screen followin the same rules as for `Form` view.

The views can be updated by the selection of records on an other view inside the same board by using *`Eval()`* on the action id of the other view in the domain.

XML description

board

Each board view must start with this tag.

- `col`: see in *common-attributes-col*.

image

Same as in `Form` view.

separator

Same as in `Form` view.

label

Same as in `Form` view.

newline

Same as in `Form` view.

notebook

Same as in `Form` view.

page

Same as in `Form` view.

group

Same as in `Form` view.

hpaned, vpaned

Same as in `Form` view.

child

Same as in `Form` view.

action

- `name`: The id of the action window.
- `colspan`: see in *common-attributes-colspan*.

Calendar view

The RNG that describes the xml for a calendar view is stored in trytond/ir/ui/calendar.rng. There is also a RNC in trytond/ir/ui/calendar.rnc.

Calendar view is use to display records as events on a calendar based on a *dtstart* and optionally a *dtend*.

XML description

calendar

Each calendar view must start with this tag.

- `dtstart`: The name of the field that contains the start date.
- `dtend`: The name of the field that contains the end date.
- `mode`: An optional name for the view that will be used first. Available views are: *day*, *week* and *month*. The default value is *month*.
- `editable`: A boolean to specify if the calendar is editable. The default value is *True*.
- `color`: An optional field name that contains the text color for the event. The default value is *black*.
- `background_color`: An optional field name that contains the background color for the event. The default value is *lightblue*.
- `width`: The minimum width the calendar should request, use -1 to unset.
- `height`: The minimum height the calendar should request, use -1 to unset.

field

- `name`: The name of the field.

Example

```
<calendar dtstart="planned_date">
    <field name="code"/>
    <field name="product"/>
    <field name="reference"/>
</calendar>
```

Extending View

Extending a view means, that the original view will be modified by a set of rules which are defined with XML.

For this purpose, the extension engine uses xpath expressions.

The view is defined with the field `inherit` of the ir.ui.view.

If the field `domain` (a Python string representation of a *domain*) is not set or evaluated to True, the inheritance will be proceeded.

XML Description

data

Each view must start with this tag.

xpath

- `expr`: the xpath expression to find a node in the inherited view.
- `position`: Define the position in relation to the node found. It can be `before`, `after`, `replace`, `inside` or `replace_attributes` which will change the attributes.

Example

```
<data>
    <xpath
        expr="/form/notebook/page/separator[@name=&quot;signature&quot;]"
        position="before">
        <label name="main_company"/>
        <field name="main_company"/>
        <label name="company"/>
        <field name="company"/>
        <label name="employee"/>
        <field name="employee"/>
    </xpath>
</data>
```

Wizard

A wizard describes a series of steps defined as *trytond.wizard.State*. The wizard stores data in *ir.session.wizard* between states.

The basics:

- Each wizard is a Python class that subclasses *trytond.wizard.Wizard*.
- The states of the wizard are attributes that are instances of *trytond.wizard.State*.

Example

This example defines a wizard which export translations

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

class TranslationExport(Wizard):
    "Export translation"
    __name__ = "ir.translation.export"

    start = StateView('ir.translation.export.start',
        'ir.translation_export_start_view_form', [
            Button('Cancel', 'end', 'tryton-cancel'),
            Button('Export', 'export', 'tryton-ok', default=True),
            ])
    export = StateTransition()
    result = StateView('ir.translation.export.result',
        'ir.translation_export_result_view_form', [
            Button('Close', 'end', 'tryton-close'),
            ])

    def transition_export(self):
        pool = Pool()
        translation_obj = pool.get('ir.translation')
        file_data = translation_obj.translation_export(
            self.start.language.code, self.start.module.name)
        self.result.file = buffer(file_data)
        return 'result'

    def default_result(self, fields):
        return {
            'file': self.result.file,
            }

Pool.register(TranslationExport, type_='wizard')
```

The class must be registered in the *Pool*.

Reports

Tryton can generate dynamic reports in many formats from templates. The reports are generated in one step as follows: a report template in a special file format, explained later, is interpolated with dynamic data and placed into a document of the same file format. Tryton's ability to generate documents in this way allows documents to be generated for any editor that supports the Open Document Format which can be converted to third party formats, such as PDF. LibreOffice must be installed on the server host for format conversion.

Report Templates

Report templates are files with a format supported by relatorio, that contain snippets of the Genshi templating language.

Here is an example of the text that would be placed in an open document text document, `*.odt`, that displays the full name and the address lines of the first address of each party. The genshi code is placed in the template using `Functions->Placeholder->Text` Fields. These are specific to ODT files.

Report API

Python API

XML Description

When defining an *ir.action.report* the following attributes are available:

- `name`: The name of the report.
- `report_name`: The name of the report model, for example my_module.my_report. This is the name you would use with *Pool().get*
- `model`: If this report is of an existing model this is its name. For example my_module.my_model. Custom reports that aren't of a specific model will need to leave this blank.
- `report`: The path to the template file starting with the module, for example my_module/my_report.odt.
- `template_extension`: The template format.
- `single`: *True* if the template works only for one record. If such report is called with more than one record, a zip file containing all the reports will be generated.

Report Usage

Using genshi and open office reports

Setting up an ODT file

If you are creating a report from scratch you should perform the following steps:

- Remove user data
  - "File > Properties..."
  - Uncheck "Apply user data"
  - Click on "Reset"
- Select Style and Formatting
  - Press F11 or "Format > Style and Formatting"
  - Click on the drop down at the right top
  - Select "Load Styles"
  - Click on "From File..."
  - Select a existing report (`company/header_A4.odt`)
- Set some parameters
  - Set the zoom to 100% (View>Zoom)
  - Set the document in read-only mode (`File>Properties>Security`) (Decreases the time it takes to open the document.)
- Usage
  - Use Liberation fonts (Only necessary if being officially included in Tryton)

- Try to use styles in report templates so that they can be extended.

Using Genshi in an ODT file

The genshi code is placed in the template using Functions->Placeholder->Text Fields. These are specific to `*.odt` files and can be found in the open office menu at Insert -> Fields -> Other and then Functions -> Placeholder -> Text. Type genshi code into the Placeholder field. There are alternatives for embedding genshi that are supported by relatorio but their use is not encouraged within Tryton.

Also note that relatorio only supports a subset of genshi. The directives that are supported by relatorio can be found here: Quick Example .

See genshi's documentation for more information: Genshi XML Templates

Examples

The modules company, account_invoice and stock all contain helpful examples.

Also see relatorio's site for some examples:

- Quick Example
- In Depth Introduction
- Example Documents

Accessing models from within the report

By default instances of the models the report is for are passed in to the report via a list of objects called *records* (or *record* if *single* is *True*). These records behave just as they would within trytond itself. You can access any of the models relations as well. For example within the invoice report each object is an invoice and you can access the name of the party of the invoice via *invoice.party.name*. Additional objects can be passed to a report. This is discussed below in *Passing custom data to a report*.

Within Tryton the underlying model the report can be found by following the Menu to `Administration > UI > Actions > Report`. Furthermore in tryton the fields for that model can be found by following the menu to `Administration > Model > Model`. Model relation fields can be accessed to any depth, for example, one could access *invoice.party.addresses* to get a list of addresses for the party of an invoice.

Creating a simple report template for a model from client

Once you have created a report template it has to be uploaded to trytond. This can be done by creating a new record in the `Administration > UI > Actions > Report` menu. Just make sure to include the template file in the content field.

In order to make the report printable from a model create a "Print form" keyword related to the model where the report should be available.

Creating a simple report template for a model in XML

Less work has to be done if you just want a simple report representation of a model. There are just 2 steps. First, create a report template file in a format supported by relatorio. Second, describe your report in XML making sure to define the correct `report_name` and `model`.

Replacing existing Tryton reports

To replace an existing report you must deactivate the old report and activate the new report.

For example to deactivate the sale report:

```
<record model="ir.action.report" id="sale.report_sale">
  <field name="active" eval="False"/>
</record>
```

Then you must activate the new sale report that exists in your new module:

```
<record model="ir.action.report" id="report_sale">
  <field name="name">Sale</field>
  <field name="report_name">sale.sale</field>
  <field name="model">sale.sale</field>
  <field name="report">my_module/sale.odt</field>
  <field name="template_extension">odt</field>
</record>
```

And create the keyword for the new report:

```
<record model="ir.action.keyword" id="report_sale_keyword">
    <field name="keyword">form_print</field>
    <field name="model">sale.sale,-1</field>
    <field name="action" ref="report_sale"/>
</record>
```

Passing custom data to a report

In this example *Report.get_context* is overridden and an employee object is set into context. Now the invoice report will be able to access the employee object.

```
from trytond.report import Report
from tryton.pool import Pool

class InvoiceReport(Report):
    __name__ = 'account.invoice'

    @classmethod
    def get_context(cls, records, data):
        pool = Pool()
        Employee = pool.get('company.employee')

        context = super(InvoiceReport, cls).get_context(records, data)
        employee_id = Transaction().context.get('employee')
        employee = Employee(employee_id) if employee_id else None
        context['employee'] = employee

        return context

Pool.register(InvoiceReport, type_='report')
```

Remote Procedure Call

There are two protocols supported by trytond: JSON-RPC (Version 1.0) and XML-RPC. The URL of the calls must end with the database name with a trailing '/'.

The available methods are:

common.db.login

It takes as parameters: the user name and a dictionary of login parameters. It returns in case of success the user ID and the session. If the parameters are not valid to authenticate the user, it returns nothing. Otherwise if it misses a key in the parameters, it raises a *LoginException* exception with the missing key name, type and the message to ask to the user.

common.db.logout

It takes no parameters and it invalidate the current session.

TODO

other methods

Authorization

Most of the calls require authorization, there are two methods:

Basic

It follows the Basic access authentication.

Session

The authorization field is constructed by the username, the user ID and the session combined with a single colon and encoded in Base64. The session is retrieved by calling the method *common.db.login*.

Task Queue

Tryton provides a way to run asynchronously some tasks. You must activate the worker in the *queue* section of the *configuration* and *run the worker manager* otherwise the tasks will be run at the end of the transaction.

A task is the parameters that defines how to call a method from a `trytond.model.Model`. This include the `trytond.transaction.Transaction.context`, the `trytond.transaction.Transaction.user` and the arguments. The first argument of the method must be an instance or a list of instances of `trytond.model.Model`. This other arguments must be JSON-ifiable.

A task is pushed into the *queue* by calling the desired method on the `trytond.model.Model.__queue__`. This stores in the queue all the current parameters of the call and it will be execute by a worker or at the end of the

transaction if no worker is configured. The following *trytond.transaction.Transaction.context* keys are used as parameters for the queue:

- *queue_name*: The name of the queue. Default value is *default*.
- *queue_scheduled_at*: A *datetime.timedelta* to add to current time to define when the task should be started. Default value is *None* which means directly.
- *queue_expected_at*: A *datetime.timedelta* to add to current time to define when the task should be finished. Default value is *None* which means as soon as possible.

Warning: There is no access right verification during the execution of the task.

Example:

```
from trytond.model import Model

class MyModel(Model):
    "My Model"
    __name__ = 'my_model'

    @classmethod
    def launch(cls, records):
        for record in records:
            cls.__queue__.process(record, 42)

    def process(self, value):
        self.value = value
```

Scheduled Actions

Tryton provides a scheduler (aka cron) which can execute methods of *models* periodically at set intervals.

The planning is managed by `ir.cron` records which store the method to call and the interval of time between calls. The method must be a class method of a *Model* which can be called without any parameters.

To register a new method with the scheduler, you must extend the `ir.cron` model and append the new method to the *selection* attribute of the `method` field in *__setup__()*. The name of the selection must be the model name and the method name joined together with a `|` between them.

Example:

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

class Cron(metaclass=PoolMeta):
    __name__ = 'ir.cron'

    @classmethod
    def __setup__(cls):
        super().__setup__()
        cls.method.selection.append(
            ('my_model|my_method', "Run my method"),
            )
```

(continues on next page)

(continued from previous page)

```
class MyModel(Model):
    "My Model"
    __name__ = 'my_model'

    @classmethod
    def my_method(cls):
        pass
```

User Application

Tryton provides a way to connect URL rules to an callable endpoint using the decorator method *route* of the *trytond.application.app* instance. This allows to define a custom API based on HTTP that can be used to create a specific user application.

The decorator takes as first parameter a string which follow the Rule Format of Werkzeug and as second parameter sequence of HTTP methods.

Example:

```
from trytond.application import app

@app.route('/hello', methods=['GET'])
def hello(request):
    return 'Hello world'
```

The following converter is added by Tryton:

- *base64*: This converter accepts any Base64 string and transforms it into its corresponding bytes value.

Tryton also provides some wrappers in *trytond.protocols.wrappers* to ease the creation of such route.

- *set_max_request_size(size)*: change the default limit of the request to the size in bytes.
- *with_pool*: which takes the first parameter as database name and replace it by the corresponding instance of the *Pool*.
- *with_transaction([readonly])*: which starts a `Transaction` using the *Pool* from *with_pool*. If *readonly* is not set, the transaction will not be readonly for *POST*, *PUT*, *DELETE* and *PATCH* methods and readonly for all others.
- *user_application(name[, json])*: which set the `Transaction.user` from the *Authorization* header using the type *bearer* and a valid key for the named user application.

User Application Key

Tryton also provides a easy way to manage access to user application using keys per named application. A key is created with a *POST* request on the *URL /<database_name>/user/application/* which returns the key. The request must contain as data a json object with the keys:

- *user*: the user login
- *application*: the name of the application

After the creation, the key must be validated by the user from the preferences of a Tryton client.

A key can be deleted with a *DELETE* request on the same *URL*. The request must contain as data a json object with the keys:

- *user*: the user login
- *key*: the key to delete
- *application*: the name of the application of the key

Sending notifications

Tryton embeds a bus system allowing the system to send text messages to clients logged in the system.

It allows the server to warn quickly the client user about some events using the *trytond.bus.notify()* function. Sending the notifications is done in a transactional way and will occur at then end of the transaction.

For example, we warn the user of low stock level when selecting a product:

```
from trytond.bus import notify

class SaleLine:
    __name__ = 'sale.line'

    def on_change_product(self):
        super().on_change_product()

        # compute the product current stock
        stock = â€¦

        if stock < 0:
            notify('Not enough stock', priority=3)
```

Modules

The modules of Tryton extend the functionality of the platform. The server comes by default with only a basic functionality included in these modules: `ir`, `res`.

Module Structure

A module is a directory in trytond/modules which contains at least two files:

- `__init__.py`: a Tryton module must be a Python module.
- `tryton.cfg`: a Configuration file that describes the Tryton module.

`__init__.py` file

It is the Python `__init__.py` to define a module. It must contains a method named *register()* that must register to the pool all the objects of the module.

`tryton.cfg` file

It is a configuration file using the format of ConfigParser that must contain *tryton* section with this following name:

- `version`: The version number of the module.
- `depends`: A one per line list of modules on which this module depends.

- `extras_depend`: A one per line list of modules on which this module *may* depend.
- `xml`: The one per line list of the XML files of the module. They will be loaded in the given order at the installation or update of the module.

Here is an example:

```
[tryton]
version=0.0.1
depends:
    ir
    res
    country
xml:
    party.xml
    category.xml
    address.xml
    contact_mechanism.xml
```

Python Files

The Python files define the models for the modules.

XML Files

The XML files define data that will be inserted into the database.

There is an rnc for those files stored in `trytond/tryton.rnc`.

The following snippet gives a first idea of what an xml file looks:

```
<?xml version="1.0"?>
<tryton>
    <data>
        <record model="res.group" id="group_party_admin">
            <field name="name">Party Administration</field>
        </record>
        <record model="res.user-res.group"
            id="user_admin_group_party_admin">
            <field name="user" ref="res.user_admin"/>
            <field name="group" ref="group_party_admin"/>
        </record>

        <menuitem name="Party Management" sequence="0" id="menu_party"
            icon="tryton-users"/>

        <record model="ir.ui.view" id="party_view_tree">
            <field name="model">party.party</field>
            <field name="type">tree</field>
            <field name="arch">
                <![CDATA[
                <tree string="Parties">
                    <field name="code"/>
                    <field name="name"/>
                    <field name="lang"/>
                    <field name="vat_code"/>
```

(continues on next page)

(continued from previous page)

```
                    <field name="active" tree_invisible="1"/>
                    <field name="vat_country" tree_invisible="1"/>
                    <field name="vat_number" tree_invisible="1"/>
                </tree>
                ]]>
            </field>
        </record>
    </data>
</tryton>
```

Here is the list of the tags:

- `tryton`: The main tag of the xml
- `data`: Define a set of data inside the file. It can have the attributes:
  - `noupdate` to prevent the framework to update the records,
  - `depends` to import data only if all modules in the comma separated module list value are installed,
  - `grouped` to create records at the end with a grouped call.
- `record`: Create a record of the model defined by the attribute `model` in the database. The `id` attribute can be used to refer to the record later in any xml file.
- `field`: Set the value of the field with the name defined by the attribute `name`.

  Here is the list of attributes:
  - `search`: Only for relation field. It contains a domain which is used to search for the value to use. The first value found will be used.
  - `ref`: Only for relation field. It contains an xml id of the relation to use as value. It must be prefixed by the module name with an ending dot, if the record is defined in an other module.
  - `eval`: Python code to evaluate and use result as value. The following expressions are available:
    - `time`: The python time module
    - `version`: The current Tryton version
    - `ref`: A function that converts an XML id into a database id.
    - `Decimal`: The python Decimal object
    - `datetime`: The python datetime module
  - `pyson`: convert the evaluated value into *PYSON* string.
  - `depends`: set value only if all modules in the comma separated module list value are installed.

  Note: Field content is considered as a string. So for fields that require other types, it is required to use the `eval` attribute.

- `menuitem`: Shortcut to create ir.ui.menu records.

  Here is the list of attributes:
  - `id`: The id of the menu.
  - `name`: The name of the menu.
  - `icon`: The icon of the menu.

- `sequence`: The sequence value used to order the menu entries.
- `parent`: The xml id of the parent menu.
- `action`: The xml id of the action linked to the menu.
- `groups`: A list of xml id of group, that have access to the menu, separated by commas.
- `active`: A boolean telling if the menu is active or not.

Translation

The translation of the user interface is provided module-wise. Translations are stored in the `locale/` directory of a module, each language in a PO-file. The official language files are named after the POSIX locale standard, e.g. de_DE.po, es.po, es_AR.po, es_EC.po...

The names of custom language files must match the code of the language in the Model ir.lang.

If a language is set `translatable`, the translations will be loaded into the database on each trytond module update.

Tryton supports derivative translations. This means that if the translation of a term is missing in one language, it will search on the parent languages. Also when activate a children language, you must also activate all parents.

Translation Wizards

Set Translations

The wizard adds new translations to the base language `en`.

Clean Translations

The wizard deletes obsolete translations from the database.

Synchronize Translations

The wizard updates the translations of the selected language based on the translations of the base language `en`. It will also remove duplicate translations with its direct parent.

Export Translations

The wizard requires to select a language and a module and will export the translations for this selection into a PO-file.

Override translations

Translations of a module can be overridden by another module. This can be done by putting a PO file into the `locale/override` directory of the module that shall contain the translations to override.

To override the translation of another module the `msgctxt` string must have the following content:

type:name:module.xml_id

- `type`: Value of the field type of ir.translation.

- `name`: Value of the field name of ir.translation.
- `module`: Value of the field module ir.translation.
- `xml_id`: The XML id that is stored in ir.model.data as fs_id

The xml_id part is optional and can be omitted if it is None.

Testing

Tryton supports both functional and unit tests.

Testing your module

Functional tests

Functional tests are written as doctests using proteus.

Unit tests

Tryton provides the `ModuleTestCase` class that bundles a set of tests that are useful for every module.

Unit tests in *ModuleTestCase* can be decorated with *trytond.tests.test_tryton.with_transaction()* to run the test in a transaction.

To use it in your own module you just have to inherit from `ModuleTestCase` and set the class attribute `module` to the name of your module.

```
from trytond.tests.test_tryton import ModuleTestCase, with_transaction

class MyModuleTestCase(ModuleTestCase):
    "My Module Test Case"
    module = 'my_module'

    @with_transaction()
    def test_method(self):
        "Test method"
        self.assertTrue(True)
```

Tests from this modules are found by the function *trytond.modules.my_module.tests.suite* which must return a *unittest.TestSuite* containing all the module's tests. This function is called by the Tryton test runner script to gather all the tests.

A typical *suite()* function thus looks like this:

```
def suite():
    suite = trytond.tests.test_tryton.suite()
    suite.addTests(unittest.TestLoader().loadTestsFromTestCase(
        MyModuleTestCase))
    suite.addTests(doctest.DocFileSuite('scenario_my_module.rst',
            tearDown=doctest_teardown, encoding='utf-8',
            optionflags=doctest.REPORT_ONLY_FIRST_FAILURE,
            checker=doctest_checker))
    return suite
```

Running your module's tests

Tryton provides the script *trytond/tests/run-tests.py*, just invoke it like that:

```
run-tests.py -m my_module
```

Testing trytond

Extending tests

Python modules extending tryton core can define additional tests that should be added to the existing ones.

Those modules must create an entry point *trytond.tests*. Any file in the module path specified by this entry point starting with *test_* and ending by *.py* will be imported. Each of those file must define a *suite()* function that returns a *unittest.TestSuite* that will be included in the trytond test suite. If the module from the entry point defines a *register* function it will be called when registering the test-specific models in the `trytond.pool.Pool`.

Running trytond tests

You should use the script *trytond/tests/run-tests.py* by invoking it like that:

```
run-tests.py [-c configuration]
```

You can use a different configuration file to check trytond against different backend.

API Reference

Models

Model API reference.

Model

class trytond.model.**Model**([*id*[, ***kwargs*]])

This is the base class that every kind of *model* inherits. It defines common attributes of all models.

Class attributes are:

Model.**__name__**
: It contains the a unique name to reference the model throughout the platform.

Model.**__rpc__**
: It contains a dictionary with method name as key and an instance of `trytond.rpc.RPC` as value.

Model.**_rec_name**
: It contains the name of the field used as name of records. The default value is 'name'.

Model.**id**
: The definition of the field `id` of records.

Model.**__queue__**
: It returns a queue caller for the model. The called method will be pushed into the queue.

Class methods:

classmethod Model.**__setup__**()
Setup the class before adding into the *trytond.pool.Pool*.

classmethod Model.**__post_setup__**()
Setup the class after added into the *trytond.pool.Pool*.

classmethod Model.**__register__**(*module_name*)
Registers the model in `ir.model` and `ir.model.field`.

classmethod Model.**default_get**(*fields_names*[, *with_rec_name*])
Returns a dictionary with the default values for each field in `fields_names`. Default values are defined by the returned value of each instance method with the pattern `default_`field_name`()`. `with_rec_name` allow to add *rec_name* value for each many2one field. The *default_rec_name* key in the context can be used to define the value of the *Model._rec_name* field.

classmethod Model.**fields_get**([*fields_names*[, *level*]])
Return the definition of each field on the model. The `level` defines the number of relations to include in the relation field definition.

classmethod Model.**__names__**([*field*])
Returns a dictionary with the name of the *model* and the *field*. It is a convenience-method used to format messages which should include those names.

Instance methods:

Model.**on_change**(*fieldnames*)
Returns the list of changes by calling *on_change* method of each field.

Model.**on_change_with**(*fieldnames*)
Returns the new values of all fields by calling *on_change_with* method of each field.

Model.**pre_validate**()
This method is called by the client to validate the instance.

ModelView

class trytond.model.**ModelView**

It adds requirements to display a view of the model in the client.

Class attributes:

ModelView.**_buttons**
It contains a dictionary with button name as key and the states dictionary for the button. This states dictionary will be used to generate the views containing the button.

Static methods:

static ModelView.**button**()
Decorate button method to check group access and rule.

static ModelView.**button_action**(*action*)
Same as *ModelView.button()* but return the action id of the XML *id* action or the action value updated by the returned value of the method.

static ModelView.**button_change**([**fields*])
Same as *ModelView.button()* but for button that change values of the fields on client side (similar to *on_change*).

> **Warning:** Only on instance methods.

Class methods:

classmethod ModelView.**fields_view_get**([*view_id*[, *view_type*[, *level*]]])

Return a view definition used by the client. The definition is:

```
{
    'model': model name,
    'type': view type,
    'view_id': view id,
    'arch': XML description,
    'fields': {
        field name: {
            ...
        },
    },
    'field_childs': field for tree,
}
```

classmethod ModelView.**view_toolbar_get**()

Returns the model specific actions and exports in a dictionary with keys:

- *print*: a list of available reports
- *action*: a list of available actions
- *relate*: a list of available relations
- *exports*: a list of available exports

classmethod ModelView.**view_attributes**()

Returns a list of XPath, attribute, value and an optional depends list. Each element from the XPath will get the attribute set with the JSON encoded value. If the depends list is set its fields are added to the view if the xpath matches at least one element.

ModelStorage

class trytond.model.**ModelStorage**

It adds storage capability.

Class attributes are:

ModelStorage.**create_uid**

The definition of the *trytond.model.fields.Many2One* field *create_uid* of records. It contains the id of the user who creates the record.

ModelStorage.**create_date**

The definition of the *trytond.model.fields.DateTime* field *create_date* of records. It contains the datetime of the creation of the record.

ModelStorage.**write_uid**

The definition of the *trytond.model.fields.Many2One* field *write_uid* of the records. It contains the id of the last user who writes on the record.

ModelStorage.**write_date**
The definition of the *trytond.model.fields.DateTime* field *write_date* of the records. It contains the datetime of the last write on the record.

ModelStorage.**rec_name**
The definition of the *trytond.model.fields.Function* field *rec_name*. It is used in the client to display the records with a single string.

Static methods:

static ModelStorage.**default_create_uid**()
Return the default value for *create_uid*.

static ModelStorage.**default_create_date**()
Return the default value for *create_date*.

Class methods:

classmethod ModelStorage.**create**(*vlist*)
Create records. `vlist` is list of dictionaries with fields names as key and created values as value and return the list of new instances.

classmethod ModelStorage.**trigger_create**(*records*)
Trigger create actions. It will call actions defined in `ir.trigger` if `on_create` is set and `condition` is true.

classmethod ModelStorage.**read**(*ids*, *fields_names*)
Return a list of dictionary for the record ids. The dictionary is composed of the fields as key and their values. `fields_names` can contain dereferenced fields from related models. Their values will be returned under the referencing field suffixed by a .. The number of *dots* in the name is not limited. The order of the returned list is not guaranteed.

classmethod ModelStorage.**write**(*records*, *values*[[, *records*, *values*], ...])
Write `values` on the list of records. `values` is a dictionary with fields names as key and writen values as value.

classmethod ModelStorage.**trigger_write_get_eligibles**(*records*)
Return eligible records for write actions by triggers. This dictionary is to pass to *trigger_write()*.

classmethod ModelStorage.**trigger_write**(*eligibles*)
Trigger write actions. It will call actions defined in `ir.trigger` if `on_write` is set and `condition` was false before *write()* and true after.

classmethod ModelStorage.**index_set_field**(*name*)
Return the index sort order of the field set calls.

classmethod ModelStorage.**delete**(*records*)
Delete records.

classmethod ModelStorage.**trigger_delete**(*records*)
Trigger delete actions. It will call actions defined in `ir.trigger` if `on_delete` is set and `condition` is true.

classmethod ModelStorage.**copy**(*records*[, *default*])
Duplicate the records. `default` is a dictionary of default value per field name for the created records.

The values of `default` may be also callable that take a dictionary containing the fields and values of the record copied and return of the value.

The keys of `default` may use the dotted notation for the *fields.One2Many* to define the default to pass to its *copy* operation.

New records are returned following the input order.

classmethod ModelStorage.**search**(*domain*[, *offset*[, *limit*[, *order*[, *count*]]]])
Return a list of records that match the *domain*.

If *offset* or *limit* are set, the result starts at the offset and has the length of the limit.

The *order* is a list of tuples defining the order of the result:

[('field name', 'ASC'), ('other field name', 'DESC'), ...]

The first element of the tuple is a field name of the model and the second is the sort ordering as *ASC* for ascending, *DESC* for descending or empty for a default order. This second element may contain 'NULLS FIRST' or 'NULLS LAST' to sort null values before or after non-null values. If neither is specified the default behavior of the backend is used.

In case the field used is a *fields.Many2One*, it is also possible to use the dotted notation to sort on a specific field from the target record. Or for a *fields.Dict* field, the dotted notation is used to sort on the key's value.

If *count* is set to *True*, then the result is the number of records.

classmethod ModelStorage.**search_count**(*domain*)
Return the number of records that match the *domain*.

classmethod ModelStorage.**search_read**(*domain*[, *offset*[, *limit*[, *order*[, *fields_names*]]]])
Call *search()* and *read()* at once. Useful for the client to reduce the number of calls.

classmethod ModelStorage.**search_rec_name**(*name*, *clause*)
Searcher for the *trytond.model.fields.Function* field *rec_name*.

classmethod ModelStorage.**search_global**(*cls*, *text*)
Yield tuples (record, name, icon) for records matching text. It is used for the global search.

classmethod ModelStorage.**browse**(*ids*)
Return a list of record instance for the `ids`.

classmethod ModelStorage.**export_data**(*records*, *fields_names*)
Return a list of list of values for each `records`. The list of values follows `fields_names`. Relational fields are defined with `/` at any depth. Descriptor on fields are available by appending `.` and the name of the method on the field that returns the descriptor.

classmethod ModelStorage.**export_data_domain**(*domain*, *fields_names*[, *offset*[, *limit*[, *order*]]])
Call *search()* and :meth'export_data' together. Useful for the client to reduce the number of calls and the data transfered.

classmethod ModelStorage.**import_data**(*fields_names*, *data*)
Create or update records for all values in `data`. The field names of values must be defined in `fields_names`. It returns the number of imported records.

classmethod ModelStorage.**check_xml_record**(*records*, *values*)
Verify if the records are originating from XML data. It is used to prevent modification of data coming from XML files. This method must be overiden to change this behavior.

classmethod ModelStorage.**validate**(*records*)
Validate the integrity of records after creation and modification. This method must be overridden to add validation and must raise an exception if validation fails.

Dual methods:

classmethod ModelStorage.**save**(*records*)
Save the modification made on the records.

Instance methods:

ModelStorage.**resources**()
Return a dictionary with the number of attachments (*attachment_count*), notes (*note_count*) and unread note (*note_unread*).

ModelStorage.**get_rec_name**(*name*)
Getter for the *trytond.model.fields.Function* field *rec_name*.

ModelSQL

class trytond.model.**ModelSQL**

It implements *ModelStorage* for an SQL database.

Class attributes are:

ModelSQL.**_table**
The name of the database table which is mapped to the class. If not set, the value of `Model._name` is used with dots converted to underscores.

ModelSQL.**_order**
The default *order* parameter of *ModelStorage.search()* method.

ModelSQL.**_order_name**
The name of the field (or an SQL statement) on which the records must be sorted when sorting on a field refering to the model. If not set, `ModelStorage._rec_name` will be used.

ModelSQL.**_history**
If true, all changes on records will be stored in a history table.

ModelSQL.**_sql_constraints**
A list of SQL constraints that are added on the table:

[('constraint name', constraint, 'xml id'), ...]

- *constraint name* is the name of the SQL constraint in the database
- constraint is an instance of *Constraint*
- message id for *trytond.i18n.gettext()*

Class methods:

classmethod ModelSQL.**__table__**()
Return a SQL Table instance for the Model.

classmethod ModelSQL.**__table_history__**()
Return a SQL Table instance for the history of Model.

classmethod ModelSQL.**__table_handler__**([*module_name*[, *history*]])
Return a TableHandler for the Model.

classmethod ModelSQL.**table_query**()
Could be defined to use a custom SQL query instead of a table of the database. It should return a SQL FromItem.

> **Warning:** By default all CRUD operation will raise an error on models implementing this method so the create, write and delete methods may also been overriden if needed.

classmethod ModelSQL.**history_revisions**(*ids*)
Return a sorted list of all revisions for ids. The list is composed of the date, id and username of the revision.

classmethod ModelSQL.**restore_history**(*ids*, *datetime*)
Restore the record ids from history at the specified date time. Restoring a record will still generate an entry in the history table.

> **Warning:** No access rights are verified and the records are not validated.

classmethod ModelSQL.**restore_history_before**(*ids*, *datetime*)
Restore the record ids from history before the specified date time. Restoring a record will still generate an entry in the history table.

> **Warning:** No access rights are verified and the records are not validated.

classmethod ModelSQL.**search**(*domain*[, *offset*[, *limit*[, *order*[, *count*[, *query*]]]]])
Same as *ModelStorage.search()* with the additional *query* argument. If *query* is set to *True*, the the result is the SQL query.

classmethod ModelSQL.**search_domain**(*domain*[, *active_test*[, *tables*]])
Convert a *domain* into a SQL expression by returning the updated tables dictionary and a SQL expression.

Where `tables` is a nested dictionary containing the existing joins:

```
{
    None: (<Table invoice>, None),
    'party': {
        None: (<Table party>, <join_on sql expression>),
        'addresses': {
            None: (<Table address>, <join_on sql expression>),
            },
        },
    }
```

Dual methods:

classmethod ModelSQL.**lock**(*records*)
Take a lock for update on the records.

Constraint

class trytond.model.**Constraint**(*table*)

It represents a SQL constraint on a table of the database and it follows the API of the python-sql expression.

Instance attributes:

Constraint.**table**
The SQL Table on which the constraint is defined.

Check

class trytond.model.**Check**(*table*, *expression*)

It represents a check *Constraint* which enforce the validity of the expression.

Instance attributes:

Check.**expression**
The SQL expression to check.

Unique

class trytond.model.**Unique**(*table*, **columns*)

It represents a unique *Constraint* which enforce the uniqeness of the group of columns with respect to all the rows in the table.

Instance attributes:

Unique.**columns**
The tuple of SQL Column instances.

Unique.**operators**
The tuple of *Equal* operators.

Exclude

class trytond.model.**Exclude**(*table*[, *(expression*, *operator)*, ...[, *where*]])

It represents an exclude *Constraint* which guarantees that if any two rows are compared on the specified expression using the specified operator not all of these comparisons will return *TRUE*.

Instance attributes:

Exclude.**excludes**
The tuple of expression and operator.

Exclude.**columns**
The tuple of expressions.

Exclude.**operators**
The tuple of operators.

Exclude.**where**
The clause for which the exclusion applies.

Workflow

class trytond.model.**Workflow**

A Mix-in class to handle transition check.

Class attribute:

Workflow.**_transition_state**
The name of the field that will be used to check state transition.

Workflow.**_transitions**
A set containing tuples of from and to state.

Static methods:

static Workflow.**transition**(*state*)
Decorate method to filter ids for which the transition is valid and finally to update the state of the filtered ids.

ModelSingleton

class trytond.model.**ModelSingleton**

Modify *ModelStorage* into a singleton. This means that there will be only one record of this model. It is commonly used to store configuration value.

Class methods:

classmethod ModelSingleton.**get_singleton**()
Return the instance of the unique record if there is one.

DictSchemaMixin

class trytond.model.**DictSchemaMixin**

A mixin for the schema of *trytond.model.fields.Dict* field.

Class attributes are:

DictSchemaMixin.**name**
The definition of the *trytond.model.fields.Char* field for the name of the key.

DictSchemaMixin.**string**
The definition of the *trytond.model.fields.Char* field for the string of the key.

DictSchemaMixin.**help**
The definition of the *trytond.model.fields.Char* field used as the help text for the key.

DictSchemaMixin.**type_**
The definition of the *trytond.model.fields.Selection* field for the type of the key. The available types are:

- boolean
- integer
- char
- float
- numeric
- date
- datetime
- selection

DictSchemaMixin.**digits**
The definition of the *trytond.model.fields.Integer* field for the digits number when the type is *float* or *numeric*.

DictSchemaMixin.**domain**
A *domain* constraint on the dictionary key that will be enforced only on the client side.

The key must be referenced by its name in the left operator of the domain. The *PYSON* evaluation context used to compute the domain is the dictionary value. Likewise the domain is tested using the dictionary value.

DictSchemaMixin.**selection**
The definition of the *trytond.model.fields.Text* field to store the couple of key and label when the type is *selection*. The format is a key/label separated by ":" per line.

DictSchemaMixin.**selection_sorted**
If the *selection* must be sorted on label by the client.

DictSchemaMixin.**selection_json**
The definition of the *trytond.model.fields.Function* field to return the JSON version of the *selection*.

Static methods:

static DictSchemaMixin.**default_digits**()
Return the default value for *digits*.

Class methods:

classmethod DictSchemaMixin.**get_keys**(*records*)
Return the definition of the keys for the records.

classmethod DictSchemaMixin.**get_relation_fields**()
Return a dictionary with the field definition of all the keys like the result of *Model.fields_get()*.

It is possible to disable this method (returns an empty dictionary) by setting in the *dict* section of the configuration, the *Model.__name__* to `False`.

Instance methods:

DictSchemaMixin.**get_selection_json**(*name*)
Getter for the *selection_json*.

MatchMixin

class trytond.model.**MatchMixin**

A mixin to add to a *Model* a match method on pattern. The pattern is a dictionary with field name as key and the value to compare. The record matches the pattern if for all dictionary entries, the value of the record is equal or not defined.

Instance methods:

MatchMixin.**match**(*pattern*[, *match_none*])
Return if the instance match the pattern. If *match_none* is set *None* value of the instance will be compared.

UnionMixin

class trytond.model.**UnionMixin**

A mixin to create a *ModelSQL* which is the UNION of some *ModelSQL*'s. The ids of each models are sharded to be unique.

Static methods:

static UnionMixin.**union_models**()
Return the list of *ModelSQL*'s names

Class methods:

classmethod UnionMixin.**union_shard**(*column*, *model*)
Return a SQL expression that shards the column containing record id of model name.

classmethod UnionMixin.**union_unshard**(*record_id*)
Return the original instance of the record for the sharded id.

classmethod UnionMixin.**union_column**(*name*, *field*, *table*, *Model*)
Return the SQL column that corresponds to the field on the union model.

classmethod UnionMixin.**union_columns**(*model*)
Return the SQL table and columns to use for the UNION for the model name.

sequence_ordered

trytond.model.**sequence_ordered**([*field_name*[, *field_label*[, *order*]]])

Retuns a mixin class which defines the order of a *ModelSQL* with an *trytond.model.fields.Integer* field. field_name indicates the name of the field to be created and its default values is *sequence*. field_label defines the label which will be used by the field and defaults to *Sequence*. Order specifies the order direction and defaults to *ASC NULLS FIRST*.

MultiValueMixin

class trytond.model.**MultiValueMixin**

A mixin for *Model* to help having *trytond.model.fields.MultiValue* fields with multi-values on a *ValueMixin*. The values are stored by creating one record per pattern. The patterns are the same as those on *MatchMixin*.

Class methods:

classmethod MultiValueMixin.**multivalue_model**(*field*)
Return the *ValueMixin* on which the values are stored for the field name. The default is class name suffixed by the field name.

classmethod MultiValueMixin.**setter_multivalue**(*records*, *name*, *value*, ***pattern*)
The setter method for the *trytond.model.fields.Function* fields.

Instance methods:

MultiValueMixin.**multivalue_records**(*field*)
Return the list of all *ValueMixin* records linked to the instance. By default, it returns the value of the first found *trytond.model.fields.One2Many* linked to the multivalue model or all the records of this one.

MultiValueMixin.**multivalue_record**(*field*, ***pattern*)
Return a new record of *ValueMixin* linked to the instance.

MultiValueMixin.**get_multivalue**(*name*, ***pattern*)
Return the value of the field *name* for the pattern.

MultiValueMixin.**set_multivalue**(*name*, *value*[, *save*], ***pattern*)
Store the value of the field *name* for the pattern. If *save* is true, it will be stored in the database, otherwise the modified *ValueMixin* records are returned unsaved. *save* is true by default.

> **Warning:** To customize the pattern, both methods must be override the same way.

ValueMixin

class trytond.model.**ValueMixin**

A mixin to store the values of *MultiValueMixin*.

DeactivableMixin

class trytond.model.**DeactivableMixin**

A mixin to add soft deletion to the model. It renders all the fields as read-only when the record is inactive.

Class attributes are:

DictSchemaMixin.**active**
: The definition of the *trytond.model.fields.Boolean* field to store soft deletion state. False values will be consideres as soft deletion.

tree

trytond.model.**tree**([*parent*[, *name*[, *separator*]]])

Returns a mixin class *TreeMixin*. *parent* indicates the name of the field that defines the parent of the tree and its default value is *parent*. *name* indicates the name of the field that defines the name of the record and its default value is *name*. If *separator* is set, the *ModelStorage.get_rec_name()* constructs the name by concatenating each parent names using it as separator and *ModelStorage.search_rec_name()* is adapted to search across the tree.

class trytond.model.**TreeMixin**

classmethod TreeMixin.**check_recursion**(*records*)
: Helper method that checks if there is no recursion in the tree defined by *tree()*.

Fields

Fields define the behavior of the data on model's record.

Field options

The following arguments are available to all field types. All are optional except *Field.string*.

string

Field.**string**

A string for the label of the field.

help

Field.**help**

A multi-line help string for the field.

required

Field.**required**

If True, the field is not allowed to be empty. Default is False.

readonly

Field.**readonly**

If True, the field is not editable in the client. Default is False.

Warning: For relational fields, it means only the new, delete, add and remove buttons are inactivated. The editable state of the target record must be managed at the target model level.

domain

Field.**domain**

A *domain* constraint that will be applied on the field value.

states

Field.**states**

A dictionary that defines dynamic states of the field and overrides the static one. Possible keys are required, readonly and invisible. The values are *PYSON* statements that will be evaluated with the values of the record.

select

Field.**select**

If true, the content of the field will be indexed.

on_change

Field.**on_change**

A set of field names. If this attribute is set, the client will call the method on_change_<field name> of the model when the user changes the current field value and will give the values of each fields in this list. The method signature is:

```
on_change_<field name>()
```

This method must change the value of the fields to be updated.

Note: The on_change_<field name> methods are running in a rollbacked transaction.

The set of field names could be filled by using the decorator *depends()*.

on_change_with

Field.**on_change_with**

A set of field names. Same like `on_change`, but defined the other way around. If this attribute is set, the client will call the method `on_change_with_<field name>` of the model when the user changes one of the fields defined in the list and will give the values of each fields in this list. The method signature is:

```
on_change_with_<field name>()
```

This method must return the new value of the field.

Note: The on_change_with_<field name> methods are running in a rollbacked transaction.

The set of field names could be filled by using the decorator *depends()*.

depends

Field.**depends**

A list of field names on which the current one depends. This means that the client will also read these fields even if they are not defined on the view. *Field.depends* is used per example to ensure that *PYSON* statement could be evaluated.

context

Field.**context**

A dictionary which will update the current context for *relation field*.

Warning: The context could only depend on direct field of the record and without context.

loading

Field.**loading**

Define how the field must be loaded: `lazy` or `eager`.

name

Field.**name**

The name of the field.

Instance methods:

Field.**convert_domain**(*domain*, *tables*, *Model*)
: Convert the simple *domain* clause into a SQL expression or a new domain. *tables* could be updated to add new joins.

Field.**sql_format**(*value*)
: Convert the value to use as parameter of SQL queries.

Field.**sql_type**()
Return the namedtuple('SQLType', 'base type') which defines the SQL type to use for creation and casting. Or *None* if the field is not stored in the database.

sql_type is using the *_sql_type* attribute to compute its return value. The backend is responsible for the computation.

For the list of supported types by Tryton see *backend types*.

Field.**sql_cast**(*expression*)
Return the SQL expression with cast with the type of the field.

Field.**sql_column**(*table*)
Return the Column instance based on table.

Field.**set_rpc**(*model*)
Adds to *model* the default RPC instances required by the field.

Field.**definition**(*model*, *language*)
Returns a dictionary with the definition of the field.

Field.**definition_translations**(*model*, *language*)
Returns a list of translation sources used by `definition()`.

Default value

See *default value*

Searching

A class method could be defined for each field which must return a SQL expression for the given domain instead of the default one. The method signature is:

```
domain_<field name>(domain, tables)
```

Where `domain` is the simple *domain* clause and `tables` is a nested dictionary, see `convert_domain()`.

Ordering

A class method could be defined for each field which must return a list of SQL expression on which to order instead of the field. The method signature is:

```
order_<field name>(tables)
```

Where `tables` is a nested dictionary, see `convert_domain()`.

Depends

trytond.model.fields.**depends**([**fields*[, *methods*]])

A decorator to define the field names on which the decorated method depends. The *methods* argument can be used to duplicate the field names from other decorated methods. This is useful if the decorated method calls another method.

Field types

Boolean

class trytond.model.fields.**Boolean**(*string*[, ***options*])

A true/false field.

Integer

class trytond.model.fields.**Integer**(*string*[, ***options*])

An integer field.

BigInteger

class trytond.model.fields.**BigInteger**(*string*[, ***options*])

A long integer field.

Char

class trytond.model.fields.**Char**(*string*[, *size*[, *translate*[, ***options*]]])

A single line string field.

Char has two extra optional arguments:

Char.**size**
: The maximum length (in characters) of the field. The size is enforced at the storage level and in the client input.

Char.**translate**
: If true, the value of the field is translatable. The value readed and stored will depend on the `language` defined in the context.

Char.**autocomplete**
: A set of field names. If this attribute is set, the client will call the method `autocomplete_<field name>` of the model when the user changes one of those field value. The method signature is:

```
autocomplete_<field name>()
```

This method must return a list of string that will populate the ComboboxEntry in the client. The set of field names could be filled by using the decorator *depends()*.

Char.**search_unaccented**
: If this attribute is set to True, `ilike` searches will be performed on unaccented strings. The default value is True.

Warning: The database backend must supports unaccented search.

Text

class trytond.model.fields.**Text**(*string*[, *size*[, *translatable*[, ***options*]]])

A multi line string field.

Text has two extra optional arguments:

Text.**size**
: Same as *Char.size*

Text.**translate**
: Same as *Char.translate*

Float

class trytond.model.fields.**Float**(*string*[, *digits*[, ***options*]])

A floating-point number field. It will be represented in Python by a `float` instance.

Float has one extra optional arguments:

Float.**digits**
: A tuple of two integers. The first integer defines the total of numbers in the integer part. The second integer defines the total of numbers in the decimal part. Integers can be replaced by a *PYSON* statement. If digits is None or any values of the tuple is *None*, no validation on the numbers will be done.

Numeric

class trytond.model.fields.**Numeric**(*string*[, *digits*[, ***options*]])

A fixed-point number field. It will be represented in Python by a `decimal.Decimal` instance.

Numeric has one extra optional arguments:

Numeric.**digits**
: Same as *Float.digits*

Date

class trytond.model.fields.**Date**(*string*[, ***options*])

A date, represented in Python by a `datetime.date` instance.

DateTime

class trytond.model.fields.**DateTime**(*string*[, *format*, ***options*])

A date and time, represented in Python by a `datetime.datetime` instance. It is stored in UTC while displayed in the user timezone.

DateTime.**format**
: A string format as used by strftime. This format will be used to display the time part of the field. The default value is *%H:%M:%S*. The value can be replaced by a *PYSON* statement.

Timestamp

class trytond.model.fields.**Timestamp**(*string*[, ***options*])

A timestamp, represented in Python by a `datetime.datetime` instance.

Time

class trytond.model.fields.**Time**(*string*[, *format*, ***options*])

A time, represented in Python by a `datetime.time` instance.

Time.**format**
: Same as *DateTime.format*

TimeDelta

class trytond.model.fields.**TimeDelta**(*string*[, *converter*[, ***options*]])

An interval, represented in Python by a `datetime.timedelta` instance.

TimeDelta.**converter**
: The name of the context key containing the time converter. A time converter is a dictionary with the keys: `s` (second), `m` (minute), `h` (hour), `d` (day), `w` (week), `M` (month), `Y` (year) and the value in second.

Binary

class trytond.model.fields.**Binary**(*string*[, ***options*])

A binary field. It will be represented in Python by a `bytes` instance.

> **Warning:** If the context contains a key composed of the model name and field name separated by a dot and its value is the string *size* then the read value is the size instead of the content.

Binary has three extra optional arguments:

Binary.**filename**
: Name of the field that holds the data's filename. Default value is an empty string, which means the data has no filename (in this case, the filename is hidden, and the "Open" button is hidden when the widget is set to "image").

Binary.**file_id**
: Name of the field that holds the *FileStore* identifier. Default value is *None* which means the data is stored in the database. The field must be on the same table and accept *char* values.

> **Warning:** Switching from database to file-store is supported transparently. But switching from file-store to database is not supported without manually upload to the database all the files.

Binary.**store_prefix**
: The prefix to use with the *FileStore*. Default value is *None* which means the database name is used.

Selection

class trytond.model.fields.**Selection**(*selection*, *string*[, *sort*[, *selection_change_with*[, *translate*[, ***options*]]]])

A string field with limited values to choose from.

Selection has one extra required argument:

Selection.**selection**
A list of 2-tuples that looks like this:

```
[
    ('M', 'Male'),
    ('F', 'Female'),
]
```

The first element in each tuple is the actual value stored. The second element is the human-readable name.

It can also be the name of a class or instance method on the model, that will return an appropriate list. The signature of the method is:

```
selection()
```

Note: The method is automaticly added to `trytond.model.Model._rpc` if not manually set.

Selection has two extra optional arguments:

Selection.**sort**
If true, the choices will be sorted by human-readable value. Default value is `True`.

Selection.**selection_change_with**
A set of field names. If this attribute is set, the client will call the `selection` method of the model when the user changes on of the fields defined in the list and will give the values of each fields in the list. The `selection` method should be an instance method. The set of field names could be filled by using the decorator *depends()*.

Selection.**translate_selection**
If true, the human-readable values will be translated. Default value is `True`.

Instance methods:

Selection.**translated**([*name*])
Returns a descriptor for the translated value of the field. The descriptor must be used on the same class as the field. It will use the language defined in the context of the instance accessed.

MultiSelection

class trytond.model.fields.**MultiSelection**(*selection*, *string*[, *sort*[, *translate*[, ***options*]]])

A list field with limited values to choose from.

MultiSelection has one extra required argument:

MultiSelection.**selection**
Same as *Selection.selection*

MultiSelection has two extra optional arguments:

MultiSelection.**sort**
Same as *Selection.sort*

MultiSelection.**translate_selection**
Same as *Selection.translate_selection*

Instance methods:

MultiSelection.**translated**([*name*])
Same as *Selection.translated()* but returns a list of translated values.

Reference

class trytond.model.fields.**Reference**(*string*[, *selection*[, *sort*[, *selection_change_with*[, *translated*[, *search_order*[, *search_context*[, **options*]]]]]]])

A field that refers to a record of a model. It will be represented in Python by a `str` instance like this:

```
'<model name>,<record id>'
```

But a `tuple` can be used to search or set value.

Reference has three extra optional arguments:

Reference.**selection**
Same as *Selection.selection* but only for model name.

Reference.**sort**
Same as *Selection.sort*.

Reference.**selection_change_with**
Same as *Selection.selection_change_with*.

Reference.**translate_selection**
Same as *Selection.translate_selection*.

Reference.**datetime_field**
Same as *Many2One.datetime_field*

Reference.**search_order**
Same as *Many2One.search_order*

Reference.**search_context**
Same as *Many2One.search_context*

Instance methods:

Reference.**translated**([*name*])
Same as *translated()* but for the translated name of the target model.

Many2One

class trytond.model.fields.**Many2One**(*model_name, string[, left[, right[, ondelete[, datetime_field[, target_search[, search_order[, search_context[, **options]]]]]]]]*)

A many-to-one relation field.

Many2One has one extra required argument:

Many2One.**model_name**
The name of the target model.

Many2One has some extra optional arguments:

Many2One.**left**
The name of the field that stores the left value for the Modified Preorder Tree Traversal. It only works if the *model_name* is the same then the model.

Warning: The MPTT Tree will be rebuild on database update if one record is found having left or right field value equals to the default or NULL.

Many2One.**right**
The name of the field that stores the right value. See *left*.

Many2One.**ondelete**
Define the behavior of the record when the target record is deleted. Allowed values are:

- `CASCADE`: it will try to delete the record.
- `RESTRICT`: it will prevent the deletion of the target record.
- `SET NULL`: it will empty the relation field.

`SET NULL` is the default setting.

Note: `SET NULL` will be override into `RESTRICT` if *required* is true.

Many2One.**datetime_field**
If set, the target record will be read at the date defined by the datetime field name of the record. It is usually used in combination with *trytond.model.ModelSQL._history* to request a value for a given date and time on a historicized model.

Many2One.**target_search**
Define the kind of SQL query to use when searching on related target. Allowed values are:

- `subquery`: it will use a subquery based on the ids.
- `join`: it will add a join on the main query.

`join` is the default value.

Note: `join` could improve the performance if the target has a huge amount of records.

Many2One.**search_order**
A *PYSON* expression defining the default order used to display search results in the clients.

Many2One.**search_context**
A dictionary defining the default context used when searching from the client.

Beware that `search_context` will override the values from the client `context`.

One2Many

class trytond.model.fields.**One2Many**(*model_name*, *field*, *string*[, *add_remove*[, *order*[, *datetime_field*[, *size*[, *search_order*[, *search_context*[, ***options*]]]]]]])

A one-to-many relation field. It requires to have the opposite *Many2One* field or a *Reference* field defined on the target model.

This field accepts as written value a list of tuples like this:

- `('create', [{<field name>: value, ...}, ...])`: it will create new target records and link them to this one.
- `('write'[[, ids, ...], {<field name>: value, ...}, ...])`: it will write values to target ids.
- `('delete'[, ids, ...])`: it will delete the target ids.
- `('add'[, ids, ...])`: it will link the target ids to this record.
- `('remove'[, ids, ...])`: it will unlink the target ids from this record.
- `('copy', ids[, {<field name>: value, ...}])`: it will copy the target ids to this record. Optional field names and values may be added to override some of the fields of the copied records.

Note: *PYSON* statement or *Field.depends* of target records can access value of the parent record fields by prepending `_parent_` to the opposite field name and followed by the dotted notation.

One2Many has some extra required arguments:

One2Many.**model_name**
The name of the target model.

One2Many.**field**
The name of the field that handles the opposite *Many2One* or *Reference*.

One2Many has some extra optional arguments:

One2Many.**add_remove**
A *domain* to select records to add. If set, the client will allow to add/remove existing records instead of only create/delete.

One2Many.**filter**
A *domain* that is not a constraint but only a filter on the records.

Warning: Only a static domain is allowed, it cannot contain any *PYSON* statements.

One2Many.**order**
A list of tuple defining the default order of the records like for *trytond.model.ModelSQL._order*.

One2Many.**datetime_field**
Same as *Many2One.datetime_field*

One2Many.**size**
An integer or a PYSON expression denoting the maximum number of records allowed in the relation.

One2Many.**search_order**
Same as *Many2One.search_order*

One2Many.**search_context**
Same as *Many2One.search_context*

Instance methods:

One2Many.**remove**(*instance*, *records*)
Remove the target records from the instance instead of deleting them.

Many2Many

class trytond.model.fields.**Many2Many**(*relation_name*, *origin*, *target*, *string*[, *order*[, *datetime_field*[, *size*[, *search_order*[, *search_context*[, ***options*]]]]]])

A many-to-many relation field. It requires to have the opposite origin *Many2One* field or a:class:*Reference* field defined on the relation model and a *Many2One* field pointing to the target.

This field accepts as written value a list of tuples like the *One2Many*.

Many2Many has some extra required arguments:

Many2Many.**relation_name**
The name of the relation model.

Many2Many.**origin**
The name of the field that has the *Many2One* or *Reference* to the record.

Many2Many.**target**
The name of the field that has the *Many2One* to the target record.

Note: A *Many2Many* field can be used on a simple *ModelView*, like in a *Wizard*. For this, *relation_name* is set to the target model and *origin* and *target* are set to *None*.

Many2Many has some extra optional arguments:

Many2Many.**order**
Same as *One2Many.order*

Many2Many.**datetime_field**
Same as *Many2One.datetime_field*

Many2Many.**size**
An integer or a PYSON expression denoting the maximum number of records allowed in the relation.

Many2Many.**add_remove**
An alias to the domain for compatibility with the *One2Many*.

Many2Many.**filter**
Same as *One2Many.filter*

Many2Many.**search_order**
Same as *Many2One.search_order*

Many2Many.**search_context**
Same as *Many2One.search_context*

Instance methods:

Many2Many.**get_relation**()
Return the relation *Model*.

Many2Many.**get_target**()
Return the target *Model*.

Many2Many.delete(instance, records):
Delete the target records from the instance instead of removing them.

One2One

class trytond.model.fields.**One2One**(*relation_name*, *origin*, *target*, *string*[, *datetime_field*[, ***options*]])

A one-to-one relation field.

Warning: It is on the relation_name *Model* that the unicity of the couple (origin, target) must be checked.

One2One.**datetime_field**
Same as *Many2One.datetime_field*

One2One.**filter**
Same as *One2Many.filter*

Instance methods:

Many2Many.**get_target**()

Return the target *Model*.

One2One.**get_target**()
Return the target *Model*.

Function

class trytond.model.fields.**Function**(*field*, *getter*[, *setter*[, *searcher*]])

A function field can emulate any other given *field*.

Function has a required argument:

Function.**getter**
The name of the classmethod or instance of the *Model* for getting values. The signature of the classmethod is:

```
getter(instances, name)
```

where *name* is the name of the field, and it must return a dictionary with a value for each instance.

Or the signature of the classmethod is:

```
getter(instances, names)
```

where *names* is a list of name fields, and it must return a dictionary containing for each names a dictionary with a value for each instance.

The signature of the instancemethod is:

```
getter(name)
```

where *name* is the name of the field, and it must return the value.

Function has some extra optional arguments:

Function.**setter**
The name of the classmethod of the *Model* to set the value. The signature of the method id:

```
setter(instances, name, value)
```

where *name* is the name of the field and *value* the value to set.

Warning: The modifications made to instances will not be saved automatically.

Function.**searcher**
The name of the classmethod of the *Model* to search on the field. The signature of the method is:

```
searcher(name, clause)
```

where *name* is the name of the field and *clause* is a *domain clause*. It must return a list of *domain* clauses but the `operand` can be a SQL query.

Instance methods:

Function.**get**(*ids*, *model*, *name*[, *values*])
Call the *getter* classmethod where *model* is the *Model* instance of the field, *name* is the name of the field.

Function.**set**(*ids*, *model*, *name*, *value*)
Call the *setter* classmethod where *model* is the *Model* instance of the field, *name* is the name of the field, *value* is the value to set.

Function.**search**(*model*, *name*, *clause*)
Call the *searcher* classmethod where *model* is the *Model* instance of the field, *name* is the name of the field, *clause* is a clause of *domain*.

MultiValue

class trytond.model.fields.**MultiValue**(*field*)

A multivalue field that is like a *Function* field but with predefined *getter* and *setter* that use the *MultiValueMixin* for stored values.

Warning: The *get_multivalue()* and *set_multivalue()* should be prefered over the descriptors of the field.

Warning: The *default* method of the field must accept pattern as keyword argument.

Dict

class trytond.model.fields.**Dict**(*schema_model*[, ***options*])

A dictionary field with predefined keys.

Note: It is possible to store the dict as JSON in the database if the backend supports by manually altering the column type to JSON on the database.

Dict has one extra required argument:

Dict.**schema_model**
The name of the DictSchemaMixin model that stores the definition of keys.

Dict.**search_unaccented**
Same as *Char.search_unaccented* but when searching on key's value.

Instance methods:

Dict.**translated**([*name*[, *type_*]])
Returns a descriptor for the translated *values* or *keys* of the field following *type_*. The descriptor must be used on the same class as the field. Default *type_* is *values*.

Wizard

A wizard is a finite state machine.

There is also a more *practical introduction into wizards*.

class trytond.wizard.**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. If an instance method with this name exists on the wizard, it will be called on deletion of the wizard and it may return one of the *client side action keywords*.

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

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

Class methods are:

classmethod Wizard.**__setup__**()
Setup the class before adding into the *trytond.pool.Pool*.

classmethod Wizard.**__post_setup__**()
Setup the class after added into the *trytond.pool.Pool*.

classmethod Wizard.**__register__**(*module_name*)
Register the wizard.

classmethod Wizard.**create**()
Create a session for the wizard and returns a tuple containing the session id, the starting and ending state.

classmethod Wizard.**delete**(*session_id*)
Delete the session.

classmethod Wizard.**execute**(*session_id*, *data*, *state_name*)
Execute the wizard for the state. *session_id* is a session id. *data* is a dictionary with the session data to update. *active_id*, *active_ids*, *active_model* and *action_id* must be set in the context according to the records on which the wizard is run.

State

class trytond.wizard.**State**
This is the base for any wizard state.

Instance attributes are:

State.**name**
The name of the state.

StateView

class trytond.wizard.**StateView**(*model_name*, *view*, *buttons*)
A *StateView* is a state that will display a form in the client. The form is defined by the *ModelView* with the name *model_name*, the *XML* id in *view* and the *buttons*. The default value of the view can be set with a method on wizard having the same name as the state but starting with *default_*.

Instance attributes are:

StateView.**model_name**
The name of the *ModelView*.

StateView.**view**
The *XML* id of the form view.

StateView.**buttons**
The list of *Button* instances to display on the form.

Instance methods are:

StateView.**get_view**(*wizard*, *state_name*)
Returns the view definition like *fields_view_get()*.

- wizard is a *Wizard* instance
- state_name is the name of the *StateView* instance

StateView.**get_defaults**(*wizard*, *state_name*, *fields*)
Return default values for the fields.

- wizard is a *Wizard* instance
- state_name is the name of the *State*
- fields is the list of field names

StateView.**get_buttons**(*wizard*, *state_name*)
Returns button definitions of the wizard.

- wizard is a *Wizard* instance
- state_name is the name of the *StateView* instance

StateTransition

class trytond.wizard.**StateTransition**
A *StateTransition* brings the wizard to the *state* returned by the method having the same name as the state but starting with *transition_*.

StateAction

class trytond.wizard.**StateAction**(*action_id*)
A *StateAction* is a *StateTransition* which let the client launch an *ir.action*. This action definition can be customized with a method on wizard having the same name as the state but starting with *do_*.

Instance attributes are:

StateAction.**action_id**
The *XML* id of the *ir.action*.

Instance methods are:

StateAction.**get_action**()
Returns the *ir.action* definition.

StateReport

class trytond.wizard.**StateReport**(*report_name*)
A *StateReport* is a *StateAction* which find the report action by name instead of *XML* id.

Button

class trytond.wizard.**Button**(*string*, *state*[, *icon*[, *default*[, *validate*]]])
A *Button* is a single object containing the definition of a wizard button.

Instance attributes are:

Button.**string**
The label display on the button.

Button.**state**
The next state to reach if button is clicked.

Button.**icon**
The name of the icon to display on the button.

Button.**default**
A boolean to set it as default on the form.

Button.**validate**
A boolean or None. If True, validation of the form will occur, if False it won't. If the value is None the validation will occur only if the state of the button is not the wizard ending state.

PYSON

PYSON is the PYthon Statement and Object Notation.

There is also a more *practical introduction into PYSON statements*.

class trytond.pyson.**PYSON**

Base class of any PYSON statement. It is never used directly.

Instance methods:

PYSON.**pyson**()

Method that returns the internal dictionary representation of the statement.

PYSON.**types**()

Method that returns a set of all possible types which the statement can become when evaluated.

classmethod PYSON.**eval**(*dct*, *context*)

Method which returns the evaluation of the statement given in `dct` within the `context`. `dct` contains a dictionary which is the internal representation of a PYSON statement. `context` contains a dictionary with contextual values.

Encoder and Decoder

class trytond.pyson.**PYSONEncoder**

Encoder for PYSON statements into string representations.

Instance method:

PYSONEncoder.**encode**(*object*)

Returns a string representation of a given PYSON statement. `object` contains a PYSON statement.

class trytond.pyson.**PYSONDecoder**([*context*[, *noeval*]])

Decoder for string into the evaluated or not PYSON statement.

Instance method:

PYSONDecoder.**decode**(*object*)

Returns a PYSON statement evaluated or not of a given string. `object` contains a string.

Statements

The following statements can be used in *PYSON*.

class trytond.pyson.**Eval**(*value*[, *default*])

An *Eval()* object represents the PYSON `Eval()` statement for evaluations. When evaluated, it returns the value of the statement named by `value`, if defined in the evaluation context, otherwise the `default` value (empty string by default).

Note: The default value determines the type of the statement.

Note: If the `value` includes dots the value will be dereferenced. For example:

```
Eval('_parent_sale.number')
```

The `number` value of the `_parent_sale` key of the evaluation context will be returned.

---

class trytond.pyson.**Not**(*value*)

A *Not* object represents the PYSON `Not()` statement for logical negations. When evaluated, returns the boolean negation of the value of the statement named by `value`, if defined in the evaluation context. Returns an instance of itself.

class trytond.pyson.**Bool**(*value*)

A *Bool* object represents the PYSON `Bool()` statement for boolean evaluations. Returns the boolean representation of the value of the statement named by `value`.

class trytond.pyson.**And**(**statements*)

An *And* object represents the PYSON `And()` statement for logical *and* operations. Returns the result of the logical conjunction of two or more values named by the statements in the `statements` tuple.

class trytond.pyson.**Or**(**statements*)

An *Or* object represents the PYSON `Or()` statement for logical *or* operations. Returns the result of the logical disjunction of two or more values named by the statements in the `statements` tuple.

class trytond.pyson.**Equal**(*statement1*, *statement2*)

An *Equal* object represents the PYSON `Equal()` statement for equation comparisons. Returns true when a value of a statement named by `statement1` and the value of a statement named by `statement2` are equal, otherwise returns false.

class trytond.pyson.**Greater**(*statement1*, *statement2*[, *equal*])

A *Greater* object represents the PYSON `Greater()` statement for *greater-than* comparisons. Returns true when the value of the statement named by `statement1` is strictly greater than the value of the statement named by `statement2`, otherwise returns false. Is the value of the variable named by `equal` is true, then returns also true when both values of statements named by `statement1` and `statement2` are equal. In this case *Greater* works as a *greater-than or equal* operator.

Note: *None* value is replaced by *0* for the comparison.

---

class trytond.pyson.**Less**(*statement1*, *statement2*[, *equal*])

A *Less* object represents the PYSON `Less()` statement for *less-than* comparisons. Returns true when the value of the statement named by `statement1` is strictly less than the value of the statement named by `statement2`, otherwise returns false. Is the value of the variable named `equal` is true, then returns also true when both values of the statements named by `statement1` and `statement2` are equal. In this case *Less* works as a *less-than or equal* operator.

Note: *None* value is replaced by *0* for the comparison.

---

class trytond.pyson.**If**(*condition*, *then_statement*, *else_statement*)

An *If* object represents the PYSON `If()` statement for conditional flow control operations. Returns the value of the statement named by `then_statement` when the value of the statement named by `condition` evaluates true. Otherwise returns the value of the statement named by `else_statement`.

class trytond.pyson.**Get**(*obj*, *key*[, *default*])

A *Get* object represents the PYSON `Get()` statement for dictionary look-up operations and evaluation. Look up and returns the value of a key named by `key` in an object named by `obj` if defined. Otherwise returns the value of the variable named by `default`.

class trytond.pyson.**In**(*key*, *obj*)

An *In* object represents the PYSON `In()` statement for look-up dictionary or integer objects. Returns true when a list (or dictionary) object named by `obj` contains the value of the variable (or key) named by `key`. Otherwise returns false.

class trytond.pyson.**Date**([*year*[, *month*[, *day*[, *delta_years*[, *delta_month*[, *delta_days*[, *start*]]]]]]])

A *Date* object represents the PYSON `Date()` statement for date related conversions and basic calculations. Returns a date object which represents the values of arguments named by the *variables* explained below. Missing values of arguments named by `year` or `month` or `day` take their defaults from `start` or the actual date. When values of arguments named by `delta_*` are given, they are added to the values of the appropriate arguments in a date and time preserving manner.

Arguments:

year Contains a PYSON statement of type int or long.

month Contains a PYSON statement of type int or long.

day Contains a PYSON statement of type int or long.

delta_years Contains a PYSON statement of type int or long.

delta_month Contains a PYSON statement of type int or long.

delta_days Contains a PYSON statement of type int or long.

start Contains a PYSON statement of type date.

class trytond.pyson.**DateTime**([*year*[, *month*[, *day*[, *hour*[, *minute*[, *second*[, *microsecond*[, *delta_years*[, *delta_months*[, *delta_days*[, *delta_hours*[, *delta_minutes*[, *delta_seconds*[, *delta_microseconds*[, *start*]]]]]]]]]]]]]])

A *DateTime* object represents the PYSON `Date()` statement for date and time related conversions and calculations. Returns a date time object which represents the values of variables named by the *arguments* explained below. Missing values of arguments named by `year`, `month`, `day`, `hour`, `minute`, `second`, `microseconds` take their defaults from `start` or the actual date and time. When values of arguments named by `delta_*` are given, these are added to the appropriate attributes in a date and time preserving manner.

Arguments:

year Contains a PYSON statement of type int or long.

month Contains a PYSON statement of type int or long.

day Contains a PYSON statement of type int or long.

hour Contains a PYSON statement of type int or long.

minute Contains a PYSON statement of type int or long.

second Contains a PYSON statement of type int or long.

microsecond Contains a PYSON statement of type int or long.

delta_years Contains a PYSON statement of type int or long.

delta_month Contains a PYSON statement of type int or long.

delta_days Contains a PYSON statement of type int or long.

delta_hours Contains a PYSON statement of type int or long.

delta_minutes Contains a PYSON statement of type int or long.

delta_seconds Contains a PYSON statement of type int or long.

delta_microseconds Contains a PYSON statement of type int or long.

start Contains a PYSON statement of type datetime.

class trytond.pyson.**Len**(*value*)

A *Len* object represents the PYSON `Len()` statement for length of a dictionary, list or string. Returns the number of items in `value`.

class trytond.pyson.**Id**(*module*, *fs_id*)

An *Id* object represents the PYSON `Id()` statement for filesystem id evaluations. When converted into the internal dictionary, it returns the database id stored in *ir.model.data*.

Transaction

class trytond.transaction.**Transaction**

This class represents a Tryton transaction that contains thread-local parameters of a database connection. The Transaction instances are context manager that will commit or rollback the database transaction. In the event of an exception the transaction is rolled back, otherwise it is commited.

Transaction.**database**
The database.

Transaction.**readonly**

Transaction.**connection**
The database connection as defined by the PEP-0249.

Transaction.**user**
The id of the user.

Transaction.**context**

Transaction.**create_records**

Transaction.**delete_records**

Transaction.**delete**

Transaction.**trigger_records**

Transaction.**timestamp**

Transaction.**started_at**
The monotonic timestamp when the transaction started.

Transaction.**language**
The language code defines in the context.

Transaction.**counter**
Count the number of modification made in this transaction.

static Transaction.**monotonic_time**()
Return a monotonic time used to populate :attr:~Transaction.started_at.

Transaction.**start**(*database_name*, *user*[, *readonly*[, *context*[, *close*[, *autocommit*]]]])
Start a new transaction and return a context manager. The non-readonly transaction will be committed when exiting the *with* statement without exception. The other cases will be rollbacked.

Transaction.**stop**([*commit*])
Stop the transaction. If commit is True, the transaction will be committed otherwise it will be rollbacked. The context manager returned by *Transaction.start()* should be used instead of calling this method.

Transaction.**set_context**(*context*, ***kwargs*)
Update the transaction context and return a context manager. The context will be restored when exiting the *with* statement.

Transaction.**reset_context**()
Clear the transaction context and return a context manager. The context will be restored when exiting the *with* statement.

Transaction.**set_user**(*user*[, *set_context*])
Modify the user of the transaction and return a context manager. *set_context* will put the previous user id in the context to simulate the record rules. The user will be restored when exiting the *with* statement.

Transaction.**set_current_transaction**(*transaction*)
Add a specific `transaction` on the top of the transaction stack. A transaction is commited or rollbacked only when its last reference is popped from the stack.

Transaction.**new_transaction**([*autocommit*[, *readonly*]])
Create a new transaction with the same database, user and context as the original transaction and adds it to the stack of transactions.

Transaction.**commit**()
Commit the transaction and all data managers associated.

Transaction.**rollback**()
Rollback the transaction and all data managers associated.

Transaction.**join**(*datamanager*)
Register in the transaction a data manager conforming to the Two-Phase Commit protocol. More information on how to implement such data manager is available at the Zope documentation.

This method returns the registered datamanager. It could be a different yet equivalent (in term of python equality) datamanager than the one passed to the method.

Transaction.**atexit**(*func*, **args*, ***kwargs*)
Register a function to be executed upon normal transaction termination. The function can not use the current transaction because it will be already committed or rollbacked.

Exceptions

exception trytond.exceptions.**TrytonException**
The base class for all Tryton exceptions.

exception trytond.exceptions.**UserError**(*message*[, *description*[, *domain*]])
The base class for exceptions used to display an error message to users. The domain may be a 2-tuple containing a *domain* and a dictionary of field definitions used to format the domain and append to the description.

exception trytond.exceptions.**UserWarning**(*name*, *message*[, *description*])
The base class for exceptions used to display a warning message to users.

exception trytond.exceptions.**LoginExceptions**(*name*, *message*[, *type*])
The exception used to request *name* parameter for the login process.

exception trytond.exceptions.**ConcurrencyException**(*message*)
The exception raised on concurrent modification.

exception trytond.exceptions.**RateLimitException**
The exception raised when user has sent too many login requests.

exception trytond.exceptions.**MissingDependenciesException**(*missings*)
The exception raised when modules are missing.

exception trytond.model.exceptions.**AccessError**
The exception raised when trying to access a record without the rights.

exception trytond.model.exceptions.**AccessButtonError**
The exception raised when trying to execute a button without the rights.

exception trytond.model.exceptions.**ImportDataError**
The exception raises when importing data fails.

exception trytond.model.exceptions.**ValidationError**
The base class for all record validation error.

exception trytond.model.exceptions.**DomainValidationError**
The exception raised when the domain of a field is not valid.

exception trytond.model.exceptions.**RequiredValidationError**
The exception raised when a required field is empty.

exception trytond.model.exceptions.**SizeValidationError**
The exception raised when the size of a field is too big.

exception trytond.model.exceptions.**DigitsValidationError**
The exception raised when the value of a field does not respect its digits.

exception trytond.model.exceptions.**SelectionValidationError**
The exception raised when the value is not in the selection.

exception trytond.model.exceptions.**TimeFormatValidationError**
The exception raised when the time format of a field is not respected.

exception trytond.model.exceptions.**ForeignKeyError**
The exception raised when a foreign key is not respected.

exception trytond.model.exceptions.**SQLConstraintError**
The exception raised when a *_sql_constraints* is not respected.

exception trytond.model.exceptions.**RecursionError**
The exception raised by *check_recursion*.

Tools

Tools API reference.

Miscellaneous

trytond.tools.**resolve**(*name*)

Resolve a dotted name to a global object.

Singleton

class trytond.tools.singleton.**Singleton**

A class to use as metaclass to create a singleton object.

Pool

class trytond.pool.**Pool**([*database_name*])

The Pool store the instances of models, wizards and reports per database.

Static methods:

static Pool.**register**(*klass*, ..., *type_*, *module*[, *depends*])
Register the classes of type (*model*, *report* or *wizard*) for the module. If `depends` is set, the registration happens only if all the modules are activated.

static Pool.**register_mixin**(*mixin*, *classinfo*, *module*)
Register the mixin for the module. The mixin will be included to all subclasses of classinfo.

Class methods:

classmethod Pool.**start**()
Start the pool by registering all Tryton modules found.

classmethod Pool.**stop**(*database_name*)
Stop the pool by removing instances for the database.

classmethod Pool.**database_list**()
List all started database.

Instance methods:

Pool.**get**(*name*[, *type*])
Return the named instance of type from the pool.

Pool.**iterobject**([*type*])
Return an interator over instances names.

Pool.**fill**(*module*, *modules*)
Fill the pool with the registered class from the module and for the activated modules and return a list of classes for each type in a dictionary.

Pool.**setup**([*classes*])
Call all setup methods of the classes provided or for all the registered classes.

Pool.**setup_mixin**(*modules*)
Include all the mixin registered for the modules to the corresponding registered classes.

PoolMeta

class trytond.pool.**PoolMeta**

The PoolMeta is a metaclass helper to setup __name__ on class to be registered in the Pool.

PoolBase

class trytond.pool.**PoolBase**

The base class of registered class that will be setup.

RPC

class trytond.rpc.**RPC**([*readonly*[, *instantiate*[, *result*[, *check_access*[, *unique*[, *fresh_session*[, *cache*]]]]]]])

RPC is an object to define the behavior of Remote Procedure Call.

Instance attributes are:

RPC.**readonly**
The transaction mode

RPC.**instantiate**
The position or the slice of the argument to be instanciated

RPC.**result**
The function to transform the result

RPC.**check_access**
Set *_check_access* in the context to activate the access right on model and field. Default is *True*.

RPC.**unique**
If set, it ensures the instantiated records are unique. Default is *True*.

RPC.**fresh_session**
If set, it requires a fresh session. Default is *False*.

RPC.**cache**
A *RPCCache* instance to compute the cache duration for the answer.

RPCCache

class trytond.rpc.**RPCCache**(*[days[, seconds]*)

Instance attributes are:

RPC.**duration**
A timdelta instance.

Instance methods are:

RCP.**headers**()
Returns a dictionary of the headers.

Internationalization

trytond.i18n.**gettext**(*message_id*[, *language*[, ***variables*]])
Returns the message translated into the *language*. The *message_id* is the *XML* id for the *ir.message* that is to be translated, and the *variables* keyword arguments are used as a mapping to format the string. If *language* is not set, then the `Transaction.language` is used.

trytond.i18n.**lazy_gettext**(*message_id*[, *language*[, ***variables*]])
Return a LazyString that will be translated with gettext later when actually used.

Sendmail

trytond.sendmail.**sendmail_transactional**(*from_addr*, *to_addrs*, *msg*[, *transaction*[, *data-manager*]])

Send email message only if the current transaction is successfully committed. The required arguments are an RFC 822 from-address string, a list of RFC 822 to-address strings (a bare string will be treated as a list with 1 address), and an email message. The caller may pass a `Transaction` instance to join otherwise the current one will be joined. A specific data manager can be specified otherwise the default *SMTPDataManager* will be used for sending email.

Warning: An SMTP failure will be only logged without raising any exception.

trytond.sendmail.**sendmail**(*from_addr*, *to_addrs*, *msg*[, *server*])

Send email message like *sendmail_transactional()* but directly without caring about the transaction. The caller may pass a server instance from smtplib.

trytond.sendmail.**get_smtp_server**([*uri*])

Return a SMTP instance from smtplib using the *uri* or the one defined in the *email* section of the *configuration*.

class trytond.sendmail.**SMTPDataManager**([*uri*])

A *SMTPDataManager* implements a data manager which send queued email at commit. An option optional *uri* can be passed to configure the SMTP connection.

SMTPDataManager.**put**(*from_addr*, *to_addrs*, *msg*)
Queue the email message to send.

FileStore

class trytond.filestore.**FileStore**

The class is used to store and retrieve files from the directory defined in the configuration *path* of *database* section. It uses a two levels of directory composed of the 2 chars of the file hash. It is an append only storage.

trytond.filestore.**get**(*id*[, *prefix*])

Retrieve the content of the file referred by the id in the prefixed directory.

trytond.filestore.**getmany**(*ids*[, *prefix*])

Retrieve a list of contents for the sequence of ids.

trytond.filestore.**size**(*id*[, *prefix*])

Return the size of the file referred by the id in the prefixed directory.

trytond.filestore.**sizemany**(*ids*[, *prefix*])

Return a list of sizes for the sequence of ids.

trytond.filestore.**set**(*data*[, *prefix*])

Store the data in the prefixed directory and return the identifiers.

trytond.filestore.**setmany**(*data*[, *prefix*])

Store the sequence of data and return a list of identifiers.

Note: The class can be overridden by setting a fully qualified name of a alternative class defined in the configuration *class* of the *database* section.

Cache

class trytond.cache.**Cache**(*name*[, *size_limit*[, *duration*[, *context*]]])

The class is used to cache values between server requests. The *name* should be unique and it's used to identify the cache. We usually use <*class_name*>.<*content_name*> to make it unique. The *size_limit* parameter can be used to limit the number of values cached and it has 1024 as the default value. The *duration* parameter defines how long a cached value stays valid but if it is not set the value remains valid until it is cleared. And the *context* parameter is used to indicate if the cache depends on the user context and is true by default. The cache is cleaned on `Transaction` starts and resets on `Transaction` commit or rollback.

Warning: As there is no deepcopy of the values cached, they must never be mutated after being set in or retrieved from the cache.

trytond.cache.**get**(*key*[, *default*])

Retrieve the value of the key in the cache. If a *default* is specified it will be returned when the key is missing otherwise it will return *None*.

trytond.cache.**set**(*key*, *value*)

Sets the *value* of the *key* in the cache.

trytond.cache.**clear**()

Clears all the keys in the cache.

classmethod trytond.cache.**sync**(*transaction*)

Synchronize cache instances using transaction.

trytond.cache.**sync_since**(*value*)

Return *True* if the last synchronization was done before *value*.

classmethod trytond.cache.**commit**(*transaction*)

Apply cache changes from transaction.

classmethod trytond.cache.**rollback**(*transaction*)

Remove cache changes from transaction.

static trytond.cache.**drop**(*dbname*)

Drops all the caches for database *dbname*

Note: By default Tryton uses a MemoryCache, but this behaviour can be overridden by setting a fully qualified name of an alternative class defined in the configuration *class* of the *cache* section.

Bus

class trytond.bus.**Bus**

The Tryton server listens on *POST* requests on the routes matching */<database_name>/bus* and replies with JSON dictionary containing:

channel The channel on which this message has been received.

message A dictionary that is the message the client must handle. The specification of the message depends of its type.

All messages should at least content a unique identifier in the key *message_id* and their type in the key of the same name.

Client sending their request on the route must be authenticated. The request must submit a JSON dictionary containing:

last_message A value identifying the last message received by the client. This value can be *null*.

channels A list of strings denoting the channels the client is listening to.

The :class:Bus exposes two methods that are used by the framework: *publish* and *subscribe*.

classmethod `Bus.`**publish**(*channel*, *message*)
Send a message to a specific channel.

Currently implemented messages are:

- *Notifications*

classmethod `Bus.`**subscribe**(*database*, *channels*[, *last_message*])
Subscribe a user client to some *channels* of messages.

The *message_id* parameter defines the last message ID received by the client. It defaults to None when not provided.

The default implementation provides an helper method to construct the response:

classmethod `Bus.`**create_response**(*channel*, *message*)
Creates a dictionary suitable as a response from a message and a timestamp.

channel is the channel on which the message has been received.

message is the content of the message sent to the client.

Please note that the implementation relies on the fact that the order of the messages received is consistent accros different trytond instances allowing to dispatch the request to any trytond server running.

Notification

Tryton provides a shortcut to send a notification with the *notify* method.

`trytond.bus.`**notify**(*title*[, *body*[, *priority*[, *user*[, *client*]]]])
Send a text message to a user's client to be displayed using a notification popup. The meaning of *title*, *body* and *priority* is defined in *Notification message*.

If *user* is not set, the current `user` is used. Otherwise *user* is the user ID to notify.

If *client* is not set then every client of the user will receive the message. If *client* and *user* are not set, the system will send the notification to the current user client. Otherwise the notification will be sent to the client whose ID matches *client*.

Notification message

Notification messages are composed of four parts:

kind The string 'notification'.

title A string. A one-line summary of the message.

body A string. The short informative message for the user. It can span multiple lines but no markup is allowed.

priority An integer between 0 (low priority) to 3 (urgent). The notification priority on the platform supporting it.

Test

trytond.tests.test_tryton.**DB_NAME**
The name of the database to use for testing. Its value is taken from the environment variable of the same name.

trytond.tests.test_tryton.**USER**
The user id used to test the transactions

trytond.tests.test_tryton.**CONTEXT**
The context used to test the transactions

trytond.tests.test_tryton.**activate_module**(*name*)

Activates the module *name* for the tested database.

In case database does not exist and the *DB_CACHE* environment variable is set then Tryton will restore a backup found in the directory pointed by *DB_CACHE*. Otherwise it will proceed to the creation of the database and the activation of the module.

ModuleTestCase

class trytond.tests.test_tryton.**ModuleTestCase**

A subclass of unittest.TestCase that tests a Tryton module. Some tests are included to ensure that the module works properly.

This class creates a temporary database with the module activated in setUpClass and drops it in the *tearDownClass* method.

ModuleTestCase.**module**
Name of the tested module.

Helpers

trytond.tests.test_tryton.**with_transaction**(*user=1*, *context=None*)

This function returns a decorator to run a test case inside a transaction. The transaction is rolled back and the cache cleared at the end of the test.

doctest helpers

trytond.tests.test_tryton.**doctest_setup**()

A function that prepares the run of the doctest by creating a database and dropping it beforehand if necessary. This function should be used as the *setUp* parameter

Deprecated since version The: *doctest_setup* function should not be used anymore to set up *DocFileSuite*. New modules should use `activate_modules()` instead.

trytond.tests.test_tryton.**doctest_teardown**()

A function that cleans up after the run of the doctest by dropping the database. It should be used as *tearDown* parameter when creating a *DocFileSuite*.

trytond.tests.test_tryton.**doctest_checker**
A specialized doctest checker to ensure the Python 2/3 compatibility

trytond.tests.test_tryton.**suite**()

A function returning a subclass of *unittest.TestSuite* that will drop the database if it does not exist prior to the run of the tests.

Tests tools

`trytond.tests.tools.`**`activate_modules`**`(`*modules*`)`

This function is used in proteus doctests to activate a list of *modules* in the scenario.

`trytond.tests.tools.`**`set_user`**`(`*user*, *config*`)`

This function will set the user of the *config* proteus connection to *user*.

4.3.6 Indices, glossary and tables

- genindex
- modindex
- search

4.4 Tryton Scripting Client

A library to access Tryton's models like a client.

4.4.1 Example of usage

```
>>> from proteus import config, Model, Wizard, Report
```

Configuration

Configuration to connect to a sqlite memory database using trytond as module.

```
>>> config = config.set_trytond('sqlite:///:memory:')
```

Installing a module

Find the module, call the activate button and run the upgrade wizard.

```
>>> Module = Model.get('ir.module')
>>> party_module, = Module.find([('name', '=', 'party')])
>>> party_module.click('activate')
>>> Wizard('ir.module.activate_upgrade').execute('upgrade')
```

Creating a party

First instanciate a new Party:

```
>>> Party = Model.get('party.party')
>>> party = Party()
>>> party.id < 0
True
```

Fill the fields:

```
>>> party.name = 'ham'
```

Save the instance into the server:

```
>>> party.save()
>>> party.name
'ham'
>>> party.id > 0
True
```

Setting the language of the party

The language on party is a *Many2One* relation field. So it requires to get a *Model* instance as value.

```
>>> Lang = Model.get('ir.lang')
>>> en, = Lang.find([('code', '=', 'en')])
>>> party.lang = en
>>> party.save()
>>> party.lang.code
'en'
```

Creating an address for the party

Addresses are store on party with a *One2Many* field. So the new address just needs to be appended to the list *addresses*.

```
>>> address = party.addresses.new(zip='42')
>>> party.save()
>>> party.addresses #doctest: +ELLIPSIS
[proteus.Model.get('party.address')(...)]
```

Adding category to the party

Categories are linked to party with a *Many2Many* field.

So first create a category

```
>>> Category = Model.get('party.category')
>>> category = Category()
>>> category.name = 'spam'
>>> category.save()
```

Append it to categories of the party

```
>>> party.categories.append(category)
>>> party.save()
>>> party.categories #doctest: +ELLIPSIS
[proteus.Model.get('party.category')(...)]
```

Print party label

There is a label report on *Party*.

```
>>> label = Report('party.label')
```

The report is executed with a list of records and some extra data.

```
>>> type_, data, print_, name = label.execute([party], {})
```

Sorting addresses and register order

Addresses are ordered by sequence which means they can be stored following a specific order. The *set_sequence* method stores the current order.

```
>>> address = party.addresses.new(zip='69')
>>> party.save()
>>> address = party.addresses.new(zip='23')
>>> party.save()
```

Now changing the order.

```
>>> reversed_addresses = list(reversed(party.addresses))
>>> while party.addresses:
...     _ = party.addresses.pop()
>>> party.addresses.extend(reversed_addresses)
>>> party.addresses.set_sequence()
>>> party.save()
>>> party.addresses == reversed_addresses
True
```

4.5 App Modules

Financial Accounting

Reports all transactions of your chart of accounts.

Analytic Accounting

Analyze your accounts.

Commission

Commission sale's agents.

EDocument

Electronic document format.

Marketing

Communicates with your customers.

Production

Transforms raw materials into finished products.

Project

Checks the profitability of projects.

Purchase

Follows your purchases.

Sale

Follows sale orders.

Sale Subscription

Subscribes to recurring service.

Stock

Tracks goods in your warehouses.

Timesheet

Tracks time spend.

Banking

Adds bank and account.

Company

Structure of the company.

Carrier

To ship.

Country

ISO 3166 list of countries and subdivisions.

Currency

ISO 4217 list of currencies.

Customs

Defines customs duty per tariff code.

Dashboard

Customized dashboard.

Google Maps

Links address to Google map.

Party

Identify parties.

Product

Manages products.

Authentication

Extends authentication mechanism.

Notification Email

Sends email.

User Role

Manage user groups with roles.

Web

Manage web features.

4.5.1 Accounting

Account

Fundamentals for most accounting needs.

Asset

Fundamentals for most accounting needs.

Belgian

Belgian accounting.

Cash Rounding

Round cash amounts.

Credit Limit

Manages credit limit of parties.

Deposit

Supports customer deposits.

Dunning

Manages dunning on receivables.

Dunning Email

Sends dunning emails.

Dunning Fee

Adds fees to dunnings.

Dunning Letter

Prints dunning letters.

Spanish

Spanish accounting.

Europe

Common European requiremenets.

French

French accounting.

French Chorus

Sends invoices via Chorus Pro.

German

German accounting.

Invoice

Manages customer and supplier invoices.

Invoice Correction

Correct price on posted invoices.

Invoice History

Historize invoice.

Invoice Line Standalone

Supports invoice line without invoice.

Invoice Secondary Unit

Adds a secondary unit of measure.

Invoice Stock

Links invoice lines and stock moves.

Payment

Manages payments.

Payment Clearing

Uses clearing account for payments.

Payment SEPA

Genrates SEPA messages for payments.

Payment SEPA CFONB

Adds CFONB flavors to SEPA.

Payment Stripe

Receives payment from Stripe.

Product

Adds accounting on product and category.

Statement

Books bank statement, cash daybook etc.

Statement AEB43

Imports statements in AEB43 format.

Statement CODA

Imports statements in CODA format.

Statement OFX

Imports statements in OFX format.

Statement Rule

Applies rules on imported statements.

Stock Anglo-Saxon

Values stock using the anglo-saxon method.

Stock Continental

Values stock using the continental method.

Stock Landed Cost

Allocates landed cost.

Stock Landed Cost Weight

Allocates landed cost based on weight.

Tax Cash

Reports tax on cash basis.

Tax Rule Country

Applies taxes per country of origin and destination.

Account Module

The account module defines fundamentals for most of accounting needs.

The module generates minimal chart of accounts for many languages. The XML files for each language are generated from the same original XML file thanks to the localize.xsl XSLT script. The script will produce on the standard output the desired XML file. The XSLT script can be launched with the following command:

```
xsltproc --stringparam lang <lang> localize.xsl minimal_chart.xml
```

where `lang` is one of the languages.

Fiscal Year

A fiscal year aggregates a set of periods that are included between two dates. A Fiscal year can be *Open*, *Closed* or *Locked*. Closing a fiscal year will close all the corresponding periods. A *Locked* fiscal can not be re-open.

- Name: The name of the fiscal year.
- Code: The code, useful for fast data entry and searching.
- Starting and Ending Date: The dates in which the periods should be included.
- Company: The company for which the fiscal year is defined.
- State: Can be *Open*, *Closed* or *Locked*.
- Periods: The list of periods.
- Post Move Sequence: The sequence to use for numbering moves in this fiscal year.

The *Balance Non-Deferral* wizard allow to create a move that will debit/credit each non-deferral account in such way to have a balance equals to zero for the fiscal year and debit/credit a counter part account.

Period

A period is mainly defined by a Starting and an Ending date, a Fiscal Year, a Type and a State (*Open*, *Closed* or *Locked*).

The type can be *Standard* or *Adjustement*: Periods of type *Standard* on the same fiscal year can not overlap. Period of type *Adjustement* can overlap other periods and are typically used for all the accounting moves that must be created when closing a fiscal year. By default, the system uses only *Standard* period when creating moves.

Each account move must be linked to a period and a move must be created on an open period.

Account Type

The Account Type Model defines the structure of the accounting reports as balance sheet and income statement. It also defines check boxes to filter its accounts by usage.

Account

An Account is defined by these fields:

- Name
- Code
- Company
- Parent Account
- Currency
- Type: The Account Type of the account.
- Debit Type: The Account Type of the account when debit > credit.
- Start and End Date: The period for which the account can be used.
- Replaced By: the account to use after end date.
- Second currency: Force all moves for the account to have this secondary currency.
- Closed: Forbid move on the account.
- Reconcile: Allow move lines of this account to be reconciled.
- Party Required: Make party required for move lines of this account.
- Taxes: This list of tax auto-complete move with new moves lines corresponding to thoses taxes if the user create a line linked to the current account and if the journal type is *Expense* or *Revenue*.
- Note

Journal

A Journal contains the following fields:

- Name
- Code
- Active: A checkbox that allow to disable the tax.
- Type: By default take one of the following values: *General*, *Revenue*, *Expense*, *Cash*, *Situation*.

Reconcile Write Off

A reconcile write off is used to set the writeoff options when reconciling unbalanced moves. It has the following fields:

- Name

- Company
- Journal: Will be used for creating the write off move
- Credit Account and Debit Account: The accounts used for the write off move depending on the amount sign.
- Active: A checkbox that allow to disable the writeoff.

Move

A Move groups a list of Move Lines. It contains the following fields:

- Name
- Reference
- Period
- Journal
- Effective Date: The date the move was created.
- Post Date: The date the move was posted.
- State: Can be *Draft* or *Posted*. A move should be balanced before being posted. Once posted the move gets a Reference number, the lines are posted and they can not be edited anymore.
- Lines: The move lines.

Moves Line:

A Move Line is an amount of money that is credited to or debited from an account. The fields are:

- Name
- Reference
- Debit and Credit: Define the debited or credited amount. Only one field can be filled.
- Account: The account.
- Move: The move that links all the corresponding lines.
- State: Can take one of the following value:
  - *Draft*: The line is part of a non-balanced move.
  - *Valid*: The line is part of a balanced move.
  - *Posted*: The line is part of a posted move.
- Second Currency and Amount Second Currency: allow to keep track of the original amount if the underlying transaction was made in an other currency.
- Maturity Date: used for payable and receivable lines. The Maturity Date is the limit date for the payment.
- Reconciliation: Hold a reconciliation number if applicable.
- Journal, Period, Date: The values on these fields comes from the corresponding move.
- Tax Lines. Gives the distribution of the amount line on the account chart

The Move Line is displayed using the account's name eventualy surrounded by parenthesis when it is a credit line.

The *Reconcile Lines* wizard allow to link move lines of the same acount for whose the credit sum is equal to the debit sum. If the selected lines are not balanced, the wizard offer to create a write-off line with the difference to make the reconciliation.

The *Unreconcile Lines* wizard allow to do the inverse operation (but doesn't reverse other operations that could have triggered by the reconciliation).

The *Reconcile Accounts* wizard allow to process one by one each party and account for reconciliation. The wizard tries to propose the best reconciliation possible. The configuration *reconciliation_chunk* in *account* section allow to define the length of lines that is allowed to search for proposal. The default is 10.

The *Group Lines* wizard allow to group payable and/or receivable lines in order to have a unique payable or receivable line.

Tax Code

The tax codes defines a tree structure and are used to create the tax reports. They are used to collect the tax amounts and the base amounts of the invoices. The Tax Code model contains the following fields:

- Name
- Code
- Active: A checkbox that allow to disable the tax code.
- Company: The company for which the tax code is defined.
- Parent, Children: Parent and children tax codes.
- Start and End date: The period for which the tax code is reported.
- Amount: The sum of lines for the selected periods:
  - Operator: + or -
  - Tax
  - Amount: *Tax* or *Base*
  - Type: *Invoice* or *Credit*

Tax

The tax model defines taxes, how the tax amount are computed and which tax code to use when creating invoices. The Tax model is defined by the following fields:

- Name
- Description
- Group
- Active: A checkbox that allow to disable the tax code.
- Sequence
- Type: May be *Percentage*, *Fixed*, or *None* for empty tax.
- Amount: If Type is *Fixed*, defines a fix amount for the tax.
- Percentage: If Type is *Percentage*, defines the percentage of the tax.

- Update Unit Price: If checked then the unit price for further tax calculation will be increased by the amount of this tax.
- Parent, Children: Parent and children taxes
- Company: The company for which the tax is defined.
- Invoice Account: The account to use when creating move lines for invoicing with this tax, for credit on revenue or for debit on expense.
- Credit Note Account: The account to use when creating move lines for credit notes with this tax, for debit on revenue or for credit on expense

If a code field is left empty, the corresponding amounts will be ignored by the tax reports.

Tax Rule

The tax rule defines a set of rules that will be applied when computing taxes. It's composed by a name, it's kind and a list of lines. If a tax matches a tax rule line, it will be replaced by the *Substituion Tax*. The *Original Tax* will be included only if the *Keep Origin* check is checked.

Templates

The Template models (Account Template, Account Type Template, Tax Template, Tax Code Template, etc) are the equivalent of their counterparts except that they are not linked to a company. Two wizard (*Create Chart of Account from Template* and *Update Chart of Account from Template*) allow to create and update the accounts from the account templates (and consequently all other models associated to templates).

Move Template

A move template allows to configure predefined moves. A Move Template is defined by the following fields:

- Name
- Company
- Keywords: The list of keywords used in the template.
- Journal
- Date: The date of the move. It must be leaved empty for today.
- Description: The description of the move. The keyword values can be substituted using the name surrounded by braces ('{' and '}').
- Lines: The list of template lines.
- Active

A wizard to create moved base on templates is available in the *Entries* menu. The templates are also available as actions when opening a journal.

Move Template Keywords

The keywords define the values asked to user to create the move based on the template. The fields are:

- Name

- String: The label used in the wizard form.
- Sequence: The sequence used to order the fields in the wizard form.
- Type:
  - *Char*
  - *Numeric*
  - *Date*
  - *Party*
- Required
- Digits: Only for numeric keyword.

Move Line Template

- Operation: *Debit* or *Credit*
- Amount: An expression that can use any keywords to compute the amount.
- Account
- Party: Only for account that requires a party.
- Description
- Taxes: The list of template tax lines

Tax Line Template

- Amount: An expression that can use any keywords to compute the amount.
- Code: The tax code to use.
- Tax

Account Asset Module

The `account_asset` module adds the depreciation of fixed assets.

Asset

An Asset defines how an asset is depreciated. It is mainly defined by:

- Product (of type "Assets").
- Journal.
- Value, Depreciated Amount and Residual Value.
- Start and End Date.
- Depreciation Method:
  - Linear
- Frequency:

  - Monthly
  - Yearly (using fixed year of 365 days)
- Lines.

The asset can be in one of this states:

- Draft

  The depreciation lines can be created.

- Running

  The accounting moves of depreciation lines are posted.

- Closed

  The value of the asset has been completely depreciated.

A wizard "Create Assets Moves" allows to post all accounting move up to a date.

The day and the month when the move will posted are defined on the accounting configuration.

Asset Line

An Asset Line defines for a date the value to depreciate.

Belgian Account Module

The Belgian account module defines the standard chart of account.

The module generates french and dutch chart of accounts / chart of taxes. The french / dutch XML files are generated from the same original XML file thanks to the localize.xsl XSLT script. The script will produce on the standard output the desired XML file. The XSLT script can be launched with the following command:

```
xsltproc --stringparam lang <lang> localize.xsl <xml file>
```

where `lang` is either `fr` or `nl`.

Account Cash Rounding Module

The `account_cash_rounding` module allows cash amounts to be rounded using the cash rounding factor of the currency.

When the invoice has to round the lines to pay, the exceeded amount is debited or credited to the accounts defined on the accounting configuration.

Sales and invoices have the rounding activated by default based on the accounting configuration flag. Purchase use the last purchase for the supplier as the default value and transfer the value to the created invoices.

Account Credit Limit

The `account_credit_limit` module manages credit limit of parties. A "Credit Limit Amount" is added on Party. The module allows to check for a party:

- If an amount will exceed his limit.
- If a dunning has reached a limit level.

and then raises an error message or a warning if the user is part of the "Account Credit Limit" group.

Account Deposit Module

The `account_deposit` module adds support for deposit accounting. A deposit is an amount paid by the customer prior to the company providing it with services or goods. A wizard on invoice allows to recall prior deposit of the party.

Account Dunning Module

The `account_dunning` module adds dunning for receivable move lines.

Procedure

A *Procedure* defines the dunning process with an ordered list of levels. A dunning will raise from one level to an other by selecting the next level that pass the test. The Procedure is set per *Party* and a default one can be configured in the accounting configuration.

Level

A *Level* defines the parameters to reach it:

- Days: The number of overdue days.

Dunning

A *Dunning* defines the dunning level of an overdue move line. Once processed, it triggers the communication defined on the procedure level. It is mainly defined by:

- Line: The overdue move line
- Procedure: The *Procedure* followed.
- Level: The current *Level*.
- Blocked: If true, it blocks the dunning to raise.

The dunning can be in one of these states:

- Draft

  The current level is not yet processed.

- Waiting

  The current level has been processed.

- Final

  All levels have been processed.

Account Dunning Email Module

The `account_dunning_email` module sends a dunning email to the party email contact after the process of dunnings for those which are at a level with *Send Email* checked. The template of the email is defined on the level.

Account Dunning Fee Module

The `account_dunning_fee` module allows to generate accounting moves as fees when processing dunning which are at a level with a *Fee* defined.

The accounting move debit the fee amount from the same account as the due line and credit the revenue account of the product. Only one move is created per dunning and level.

Fee

A *Fee* defines the parameters to apply the fee:

- Name: The string that will be used as account move description.
- Product: The service product that represent the fee.
- Journal: The journal on which the move will be posted.
- Compute Method:
  - List Price: The list price of the product.
  - Percentage: A percentage of the due amount.

Account Dunning Letter Module

The `account_dunning_letter` module generates a dunning letter after the process of dunnings for those who are at a level with *Print on Letter* checked.

Spanish Account Module

The Spanish account module defines the following charts of account:

- Plan General Contable EspaÃ±ol 2008
- Plan Contable para PYMES 2008

The chart was published as REAL DECRETO 1514/2007 on 20th November 2007.

A wizard allows to generate the following AEAT files:

- Modelo 111
- Modelo 115
- Modelo 303

The module generates the chart of accounts for the normal and pyme charts. The XML files for each variant are generated from the same original XML file thanks to the create_variant.xsl XSLT script. The script will produce on the standard output the desired XML file. The XSLT script can be launched with the following commands:

```
xsltproc --stringparam chart <chart> create_chart.xsl account.xml
xsltproc --stringparam chart <chart> create_chart.xsl tax.xml
```

where `chart` is *normal* or *pyme*

Account Europe Module

The `account_eu` module implements common accounting requirements in Europe. It includes:

- EC Sales List (ESL)

French Account Module

The French account module defines the standard chart of account

A wizard allows to generate the FEC file for a fiscal year.

Configuration

The `account_fr` module uses the section *account_fr* to retrieve some parameters:

- *fec_opening_code*: defines the journal code for the opening balance in the FEC file. The default value is *OUV*.
- *fec_opening_name*: defines the journal name for the opening balance in the FEC file. The default value is *Balance Initiale*.
- *fec_opening_number*: defines the number of the opening balance in the FEC file. The default value is *0*.

Account French Chorus Module

The `account_fr_chorus` module allows to send invoices through the Chorus Pro portal.

If the party is checked for Chorus Pro, all posted customer invoices are queued to be sent. A cron job will send them every 15 minutes by default using the credential from the accounting configuration.

Configuration

The account_fr_chorus module uses the section *account_fr_chorus* to retrieve the path of the SSL certificates.

- *certificate*: the path to the SSL certificate.
- *privatekey*: the path to the SSL private key.

> **Warning:** The private key must be unencrypted.

German Account Module

The German account module define the SKR03 chart of account.

Account Invoice Module

The `account_invoice` module adds the invoice, payment term.

Invoice

There are 4 types of invoice: *Invoice*, *Supplier Invoice*, *Credit Note* and *Supplier Credit Note*. An invoice can be in *Draft*, *Validated*, *Posted*, *Paid* or *Canceled*.

- Company: The company for which the invoice is created.
- Tax Identifier: The tax identifier that will be printed on the invoice (By default the first tax identifier of the company).
- Number: The invoice number set on validation for supplier invoices and on posting for others using the sequence defined on the fiscalyear.
- Reference: The optional external reference of the invoice.
- Party: The party for which or from which the invoice is issued.
- Invoice Address: The address of the party.
- Party Tax Identifier: The tax identifier of the party.
- Description: An optional description of the invoice.
- Comment: A text fields to add custom comments.
- Invoice Date: The date of the invoice. It is set on posting the invoice if not.
- Accounting Date: The date to use for accounting if set otherwise it is the invoice date.
- Currency: The currency of the invoice.
- Journal: The journal on which the accounting must be booked.
- Account: The payable or receivable account.
- Payment Term: The payment term to apply for the invoice (default value comes from the party).
- Lines: The lines invoiced.
- Taxes: The taxes related to the lines.
- Untaxed, Tax, Total Amount: The amounts computed by the invoice.
- Move: The accounting move created by the invoice on validation for supplier invoices and on posting for others.
- Cancel Move: The accounting move created to cancel a posted invoice.

The *Invoice* report is stored when the invoice is posted and thus it is always the same that is returned for consistency.

A wizard allow to register a cash payment directly on the invoice. The payment could be partial or with write-off.

An other wizard allow to create a credit note from the invoice. If the option to refund is checked, the original invoice will be cancelled by the credit note.

Invoice Line

There are 4 types of lines: *Line*, *Subtotal*, *Title*, *Comment*. The *Line* are composed of:

- Product: An optional reference to the product to invoice.
- Account: The account to book the expense or revenue.
- Quantity: The quantity invoiced.
- Unit: The unit of measure in which is expressed the quantity.
- Unit Price: The unit price of the quantity in the currency of the invoice.

- Amount: The amount of the line (Unit Price multiplied by Quantity).
- Description: The description of the product or the line.
- Note: A text fields to add custom comments.
- Taxes: The taxes to apply to the amount of the line.

Invoice Tax

It groups the taxes of all the lines. The rounding of the taxes is defined in the accounting configuration and can be: *Per Document* or *Per Line*.

- Description: The description of the tax.
- Account: The account on which the tax is booked.
- Base: The base amount on which the tax is computed.
- Base Code: The *Tax Code* to record the base amount.
- Base Sign: The sign used to record the base amount on the tax code.
- Amount: The amount of the tax.
- Tax Code: The *Tax Code* to record the tax amount.
- Tax Sing: The sign used to record the tax amount on the tax code.
- Tax: The tax used for computation.
- Manual: A boolean to define manual tax (which is not linked to an invoice line).

Payment Term

It defines the maximum dates of how an due amount should be paid.

- Name: The name of the term.
- Description: The long description of the term.
- Lines:
  - Relative Deltas:
    - Day: The day of the month.
    - Month: The month of the year.
    - Day of the Week: One of the week day.
    - Months: The number of months to add.
    - Weeks: The number of weeks to add.
    - Days: The number of days to add.
  - Type:
    - *Fixed*:
      - Amount: The maximum fixed amount to pay at this date.
      - Currency: The currency of the amount.
    - *Percentage on Remainder*:

        - Ratio: The ratio to use on the remainder amount.
        - Divisor: The reversed ratio.
    * *Percentage on Total*:
        - Ratio: The ratio to use on the total amount.
        - Divisor: The reversed ratio.
    * *Remainder*

The computation of relative delta is based on the python-dateutil library. The payment term create a term for each line as far as there is still a remainder amount.

A wizard is provided to test the behaviour of the payment term. It display computed terms base on an amount and a date.

Note: The last line of payment term must be a remainder.

Payment Method

The payment options available when paying an invoice. It has the following fields:

- Name
- Company
- Journal: Will be used for creating the payment move
- Credit Account and Debit Account: The accounts used for the payment move depending on the amount sign.
- Active: A checkbox that allow to disable the payment method.

Fiscal Year Sequences

The sequence used to compute the invoice number is retrieved from the fiscalyear sequences model. At least one record must be defined for each fiscalyear. Additional criteria can be used like:

- By period

Configuration

The account_invoice module uses the section *account_invoice* to retrieve some parameters:

- *filestore*: a boolean value to store invoice report cache in the FileStore. The default value is *False*.
- *store_prefix*: the prefix to use with the FileStore. The default value is *None*.

Account Invoice Correction Module

The account invoice correction module adds a wizard on invoice which allows select lines for which the unit price must be corrected. A new invoice is created with those lines in double: once with the original quantity, once with the inverted quantity.

Account Invoice History Module

The account invoice history module activates the historization of the invoice and its related fields.

Account Invoice Line Standalone Module

The account invoice line standalone module allows to create invoice line not linked to an invoice.

Account Invoice Secondary Unit Module

The account invoice secondary unit module adds a secondary unit of measure on invoice line.

Account Invoice Stock Module

The account invoice stock module adds link between invoice lines and stock moves.

Account Payment Module

The `account_payment` module allows to generate grouped payments for receivable or payable Account Move Lines.

Payment

A Payment defines an amount to pay to a party or to receive from a party. It is mainly defined by:

- Journal.
- Kind: - Payable - Receivable
- Party.
- Line.
- Amount.
- Date.
- Description.

A payment can be created from an payable or receivable *Account Move Line* using the *Pay Lines* action. The amount is computed from the debit/credit deducing existing payments for this line.

The payment can be in one of this states:

- Draft

  The payment is waiting for approval.

- Approved

  The payment has been approved and is waiting to be processed by the wizard in a Group.

- Processing

  The payment has been processed in a Group.

- Succeeded

  The payment was successfully processed.

- Failed

  The payment was not successfully processed.

Group

A group links a set of payment of the same kind processed together inside the same journal.

Journal

A journal defines the configuration of the processing of payments.

Account Payment Clearing Module

The `account_payment_clearing` module allows to generate account move when a payment is succeeded between the receivable/payable account to a clearing account defined on the payment journal.

The clearing account will be used for statement lines which are linked to such payment.

If a delay is configured on journal for posting, a scheduled task will post automatically clearing moves after this delay.

Account Payment SEPA Module

The `account_payment_sepa` module allows to generate SEPA files for a Payment Group.

Journal

The Journal has some fields when the process method is SEPA:

- Bank Account Number.
- Payable Flavor: - pain.001.001.03 - pain.001.001.05 - pain.001.003.05
- Receivable Flavor: - pain.008.001.02 - pain.008.001.04 - pain.008.003.02
- Payable/Receivable Initiator Identifier: - SEPA Creditor Identifier - Belgian Enterprise Number - Spanish VAT Number
- Batch Booking.
- Charge Bearer: - Debtor - Creditor - Shared - Service Level

Group

The Group has a field *SEPA Messages* containing the XML messages.

Mandate

The Mandate stores information for the Direct Debit. It is mainly defined by:

- Party.
- Account Number.

- Identification.
- Type: - Recurrent - One-off
- Scheme: - CORE - Business to Business
- Signature Date

The mandate can be in one of this states:

- Draft
- Requested
- Validated
- Canceled

Message

The Message stores the incoming and outgoing XML message.

The message can be in one of this states:

- Draft
- Waiting
- Done
- Canceled

Bank to Customer Debit Credit Notification (camt.054)

For incoming message camt.054, each booked entry will succeed the corresponding payment if any return information is found. Otherwise it will fail and the return reason will be stored on it.

Party

The Party has a field *SEPA Creditor Identifier* used for the party of the company.

Configuration

The account_payment_sepa module uses the section *account_payment_sepa* to retrieve some parameters:

- *filestore*: a boolean value to store SEPA message in the FileStore. The default value is *False*.
- *store_prefix*: the prefix to use with the FileStore. The default value is *None*.

Account Payment SEPA CFONB Module

The `account_payment_sepa_cfonb` module adds CFONB flavors to SEPA messages.

Account Payment Stripe Module

The `account_payment_stripe` module allows to receive payment from Stripe. It uses Stripe.js and Stripe Elements in a checkout form to handle Setup Intent and Payment Intent by card.

Account

The Account stores the information about the Stripe account like the secret key, the publishable key and the webhook signing secret.

The account's webhook endpoint is the URL used by stripe webhooks. For additional security, the Stripe's requests signature can be verified if the webhook signing secret is set on the Account. If no webhook is setup, a cron task fetches the new events.

Customer

The Customer allow to register parties as Stripe customers. The checkout button opens the Stripe checkout form.

A cron task runs every hour to create new customers on Stripe and another to delete them if they are inactivated.

Journal

The journal has a new field for the Stripe account.

Payment

The payment has also a checkout button which opens the Stripe checkout form. A payment can be processed off-session using a source or payment method from the customer. In case the payment method requires authorization, an email is sent to the party with a link to the checkout form. In case of error, it has also new fields which display the error messages.

A cron task runs every 15 minutes to charge each processing payment.

The capture box can be unchecked to only authorize on processing and capture the amount in a second step.

If the payment is disputed, it will be updated at the closure of the dispute.

It is possible to partially or completely refund a payment.

Configuration

The account_payment_stripe module uses the section *account_payment_stripe* to retrieve some parameters:

- *sources_cache*: defines the duration in seconds the sources are kept in the cache. The default value is *15 * 60*.
- *max_network_retries*: defines the maximum number of retries the Stripe library may perform. The default value is *3*.

Account Product Module

The account product module adds accounting on product and category.

Note: If the analytic account module is installed, it also adds the product and category as criteria for the analytic rule engine.

Account Statement Module

The `account_statement` module allows to book statements. Statement can be used for bank statement, cash daybook etc.

Statement

A statement groups many lines under a journal for a date. It is mainly defined by:

- Name.
- Journal.
- Date.

The statement can be in one of this states:

- Draft

  The statement is waiting for validation

- Validated

  The statement is validated and is waiting to be posted. A move for each grouped lines is already created in draft state.

- Posted

  The statement is posted which means all the moves have been posted.

- Canceled

  The statement is canceled which means all the moves previously created have been deleted.

Line

A Statement Line is mainly defined by:

- Sequence: Use to order the lines in the statement.
- Number: The number to identify a transaction.
- Date.
- Amount.
- Party.
- Account.
- Invoice.
- Description.

- Move: The move created for this line.

Origin

The statement origin store the raw information from an external system that are imported. The origin are converted into statement lines.

Line Group

The line group represent the group of lines created at the validation of the statement. By default the lines of a statement are grouped by *Number*, *Date* and *Party*.

Journal

A Statement Journal is mainly defined by:

- Name.
- Journal (from account).
- Currency.
- Validation Type: - Balance - Amount - Number of Lines

The statements are validated following the validation type of the journal. The Balance validation requests to set the start and end balance (the start is automaticaly filled with the end balance of the last statement on the same journal) and the difference is checked against the total amount of the lines. The Amount validation requests to set the total amount to check against the total amount of the lines. The Number of Lines requests to set the number of unique lines on the statement.

Import Statement

A wizard to import statements from an external system. It creates statements with origins filled.

Configuration

The account_statement module uses the section *account_statement* to retrieve some parameters:

- *filestore*: a boolean value to store origin file in the FileStore. The default value is *False*.
- *store_prefix*: the prefix to use with the FileStore. The default value is *None*.

Read the Docs readthedocs.org

Page Not Present

Account Statement Coda Module

The Account Statement Coda module implements the import of the CODA file as statement.

Account Statement OFX Module

The Account statement OFX module implement the import of the OFX file as statement.

Account Statement Rule Module

The `account_statement_rule` module allows rules to be defined to complete statement lines from imported files. When the "Apply Rule" button is clicked on a statement, each rule is tested in order against each origin that does not have any lines until one is found that matches. Then the rule found is used to create the statement lines linked to the origin.

Rule

A rule is composed of two parts: matching criteria and lines.

Criteria

The criteria are matched with each origin of the statement:

- Company
- Journal
- Amount: Check if the amount is between two values
- Description: A regular expression to search for a match in the origin description.
- Information rules:
  - Key: the statement information key on which the rule applies
  - The matching value depending of the type of the key:
    - Boolean
    - Numeric: A range of value.
    - Char: A regular expression.
    - Selection

The regular expression can register the group names *party*, *bank_account* and *invoice* which are later used to search for a party and an invoice.

Lines

They define how to create the statement lines from the matching origin:

- **Amount: A Python expression evaluated with:**
  - *amount*: the amount of the origin.
  - *pending*: the amount from which previous lines have been deducted.
- Party
- Account

If the party is not filled in, one will be searched for using the *bank_account* or the *party* group names from the regular expressions. If the *invoice* group name appears in a regular expression, it will be used to find an invoice to link with.

Account Stock Anglo-Saxon Module

The `account_stock_anglo_saxon` module adds anglo-saxon accounting model for stock valuation.

A new field is added to Product and Category:

- Account Cost of Goods Sold: The account which is used to record cost of goods sold.

As usual, if the account is not defined on Product, it is the category one that is used.

The Account Moves of Invoices are modified for fiscal year with the account stock method set.

On Supplier Invoice, the cost of the Product at reception is debited on the Account Stock Supplier and only the difference is debited to the Account Expense. If the reception is not yet done then the cost is the amount on the invoice. The opposite is done on Supplier Credit Note.

On Invoice, the cost of the Product at delivery is credited from the Account Stock Customer and is debited to the Account Cost of Goods Sold. If the delivery is not yet done then the current cost is used. The opposite is done on Credit Note.

Account Stock Continental Module

The `account_stock_continental` module adds continental accounting model for stock valuation.

A new configuration field for accounting is added:

- Journal Stock: The journal used for stock move.

Four new fields are added to Product and Category:

- Account Stock: The account which is used to record stock value.
- Account Stock Supplier: The counter part account for supplier stock move.
- Account Stock Customer: The counter part account for customer stock move.
- Account Stock Production: The counter part account for production stock move.
- Account Stock Lost and Found: The counter part account for lost and found stock move.

As usual, if the "Use Category's accounts" is checked it is the category one that is used otherwise it is the product one.

An Account Move is created for each Stock Move done under a fiscal year with the account stock method set and for which one Stock Location has the type "Storage" and an the other has the type "Supplier", "Customer", "Production" or "Lost and Found".

If the Stock Move has a "Supplier" Location as origin, then the Account Stock of the Product is debited and the Account Stock Supplier of the Product is credited. The amount is the Unit Price of the move or the Cost Price of the Product if it uses the "fixed" method. The account move is inverted if it is the destination.

If the Stock Move has a "Customer" Location as destination, then the Account Stock of the Product is credited and the Account Stock Customer of the Product is debited. The amount is the current Cost Price of the Product. The account move is inverted if it is the origin.

When the Location has the type "Production", then the Account Stock Production is used instead of the Supplier/Customer.

When the Location has the type "Lost and Found", then the Account Stock Lost and Found is used instead of the Supplier/Customer.

Account Stock Landed Cost Module

The `account_stock_landed_cost` module allows to allocate landed cost on *Supplier Shipments* after their reception.

A new field is added to *Product*:

- *Landed Cost*: Only available for service, it allows such invoiced product to be used as a landed cost.

A new document *Landed Cost* defines how posted supplier invoice lines are allocated to many shipments.

The allocation methods are:

- *By Value*: The cost will be allocated according to the value of each line. (The value is: *Quantity * Unit *Price*)

Once posted, the *Landed Cost* updates the unit price of each incoming moves of the shipments using the cost of all the invoice lines according to the allocation method.

To update the cost price, the *Update Cost Price* wizard must be run on the affected products.

Account Stock Landed Cost Weight Module

The `account_stock_landed_cost_weight` module adds an allocation method based on weight of each line. (The *Weight* is taken from the *Product Measurements*)

Account Tax Cash Module

The `account_tax_cash` module allows to make tax report on cash basis.

The tax groups reported on cash basis are defined on the *Fiscal Year* and *Period*. They can also be defined on the supplier invoices which get the default values from the *Party*.

When a payment lines is added to an invoice, the tax lines for the proportional amount are set on the corresponding tax code for the current period. If a payment line is removed from an invoice, the reverse operation is applied.

When closing a *Period*, a warning is raised if there are still receivable/payable lines not reconciled or linked to an invoice.

> **Warning:** The invoice template may need to be adapated to include a legal notice when tax on cash basis is used. It can be tested with the *on_cash_basis* property of the *Invoice Tax*.

Account Tax Rule Country

The `account_tax_rule` module extends the tax rule to add origin and destination countries and subdivisions as criteria.

Tax Rule Line

Four criteria fields are added:

- From Country: The country of origin
- From Subdivision: The subdivision of origin
- To Country: The country of destination

- To Subdivision: The subdivision of destination

The countries are picked from the origin document:

- Sale:
  - The origin country and subdivision come from the address of the warehouse.
  - The destination country and subdivision come from the shipping address.
- Purchase:
  - The origin country and subdivision come from the invoice address.
  - The destination country and subdivision come from the address of the warehouse.
- Stock Consignment:
  - The origin country and subdivision come from the warehouse's address of the location or the delivery address for returned customer shipment.
  - The destination country and subdivision come from the warehouse's address of the location or the delivery address for customer shipment.

4.5.2 Analytic Accounting

Account

Fundamentals for analytic.

Invoice

Adds analytic on invoice.

Purchase

Adds analytic on purchase.

Sale

Adds analytic on sale.

Analytic Account Module

The analytic account module adds the fundamentals required to analyse accounting using multiple different axes.

A chart of analytic accounts can be setup for each axis.

Account

An analytic account is defined by these fields:

- Name
- Code
- Company
- Type:
  - Root: define an axis
  - View: sum amounts of children accounts

  - Normal: store analytic lines
  - Distribution: distribute linked lines between different accounts
- Parent
- Balance/Credit/Debit
- State:
  - Draft
  - Opened
  - Closed
- Note

Line

An analytic line defines the amount of money from a move line to be assigned to an analytic account. It contains the following fields:

- Debit/Credit
- Account
- Move Line
- Date

When the linked move is posted, an analytic state is calculated for each of the move lines. It is only valid if all the analytic axes have been completely filled. The incomplete lines can be found in the menu entry "Analytic Lines to Complete".

Rule

The module contains a rule engine that can automatically create analytic lines when the move is posted, but only if they do not already have analytic lines. The criteria for the rule engine are:

- Account
- Party
- Journal

Analytic Invoice Module

The analytic invoice modules allows to set analytic accounts on invoice line.

Analytic Purchase Module

The analytic purchase module allows to set analytic accounts on purchase line.

Analytic Sale Module

The analytic sale module allows to set analytic accounts on sale line.

4.5.3 Commission

Commission

Fundamentals to commission sale's agents.

Waiting

Creates waiting account moves.

Commission Module

The commission module allows to manage commission for sale's agent. A commission move is created when posting the invoice, following the agent's commission plan.

Agent

An agent links a party with a commission plan and method.

- The *Plan* is the commission plan.
- The *Account* is the payable account on which the commission will be credited.
- **The *Commission Method* defines when the commission will be due:**
  - *On Posting*: when the invoice is posted.
  - *On Payment*: when the invoice is paid.

Commission Plan

A plan contains a sets of lines that defines the formula to use to compute the commission amount. The line is selected by choosing the first that matches the criteria.

- The *Commission Product* is used to debit the commission using its expense account.

Line

- The *Formula* is a Python expression that will be evaluated with *amount* as the invoiced amount.

The criteria:

- *Product*.

Commission Waiting

The `commission_waiting` module allows to generate account move for each commission between the expense/revenue account to a waiting account defined on the agent.

Once the commission invoice is posted the waiting account is cleared with the payable/receivable account of the invoice.

4.5.4 Electronic Document

UN/CEFACT Supports UN/CEFACT format.

UNECE Adds UNECE codes.

EDocument UN/CEFACT Module

Implement electronic document from UN/CEFACT:

- Cross-Industry-Invoice (16B-CII)

EDocument UNECE Module

The module adds many codes from the UNECE:

- Recommendation NÂ°. 20 Codes for Units of Measure Used in International Trade
- 5153 Duty or tax or fee type name code
- 5305 Duty or tax or fee category code from

4.5.5 Marketing

Marketing Fundamentals for marketing.

Automation Uses scenario.

Marketing Module

The marketing module defines the fundamentals for marketing modules.

Marketing Automation Module

The marketing_automation module allows marketing actions to be automated. It is based on scenarios and activities that are executed on selected records.

Scenario

A scenario provides the entry point for records into an automated marketing campaign. Each record can only enter a scenario once. A scenario is defined by:

- Name
- Model: the type of record for the scenario (by default Party and Sale)
- Domain: used to filter records
- State:
  - Draft
  - Running
  - Stopped

A cron task runs periodically to find new records to apply the scenario to.

Activity

The activities form a scenario. They define which action should be triggered and when it should happen. The activities are organized as a tree and defined by:

- Name
- Parent
- Children
- On: event from the parent that triggers the activity:
  - E-Mail Opened
  - E-Mail Not Opened
  - E-Mail Clicked
  - E-Mail Not Clicked
- Condition: that the record must match to execute the activity
- Delay: before the action is executed
- Action: list of available actions

Actions

Send E-Mail

The activity send an e-mail to the party defined on the record. The E-mail is composed using an HTML Genshi template with *record* in the evaluation context. All <*a*> elements have their *href* replaced by a shortened version used to trigger children activities. If the *href* value is *unsubscribe*, it is replaced by the URL which allows the recipient to block their record for the scenario. A empty image is automatically added at the end of the <*body*> to track when emails are opened.

Record

It stores a reference to the records included in each scenario. If the record is blocked, no activity will be triggered for the record.

Record Activity

It stores the state of a record for an activity of the scenario.

A cron task runs periodically to execute any waiting record activity that is due.

Configuration

The marketing_automation modules uses parameters from the section:

- *[email]*:
  - *from*: The default *From* for the email.
- *[marketing]*:

- *automation_base*: The base URL without a path for the unsubscribe URL and the empty image. The default value is created using the configuration *[web] hostname*.

4.5.6 Production

Production

Fundamentals for production management.

Outsourcing

Outsources the production.

Routing

Defines routings.

Split

Split production orders.

Work

Manage work order for each production.

Work Timesheet

Track time per work.

Production Module

The production module defines basics for production management: Bill of material and production order.

Bill of Material

Bills of Material are list of products and quantities needed to produce a product. It is often shorten with BOM.

Production

A Production is mainly defined by a product, a BOM, a location, a quantity and two lists of moves:

- Inputs

  The moves between the storage location and the production location (as defined on the warehouse) for products used for production.

- Outputs

  The moves between the production location and the storage location for products produced.

The production can be in one of this states:

- Request

  The production is requested by the system.

- Draft

  Input and output moves are in draft.

- Waiting

  The production is waiting for action and all moves are still in draft.

- Assigned

  The input moves are assigned.

- Running

  The input moves are in state done.

- Done

  The output moves are in state done.

- Cancel

  All moves are cancelled.

The cost of the production is computed with the sum of the cost price of all incoming products. This cost is allocated to the output products based on the list price of each (the product without a list price are considered as waste).

A cron task runs every day and updates the cost of productions if the cost price of the incoming products has changed.

Production Outsourcing Module

The production outsourcing module allows to outsource production order per routing. When such outsourced production is set to waiting, a purchase order is created and its cost is added to the production.

To define an outsourced production, the routing must have a *Supplier*, a *Service* and its *Quantity* defined. Those values will be used to create the purchase order. The bought quantity is computed by multiplying the *Quantity* by the factor between the bill of material and the production quantity.

Production Routing Module

The production routing module defines the routings for production: Routing, Step and Operation.

Routing

Routings are a list of steps to follow with one of the BOMs to produce a product.

Step

Steps define the operation to perform at a specific point of the routing.

Operation

An operation is a generic action.

Production Split Module

The Production Split module adds on the production a wizard that allows to split it. The production is split into productions of *Quantity*. If *Count* is set, it will be split only this number of times. On occasion there can be a production with the remaining quantity.

Production Work Module

The production work module allows to manage work order for each production. It also adds in the production cost for the work cost.

Work Center

Work center are places in the warehouse where production operations are performed. They can be organized in a tree structure and each center can be linked to a category. A cost can be defined on the work center with two methods: *Per Cycle* or *Per Hour*.

Work

Works define for a production which operation to do at which work center. They also contains the number of cycles consumed to perform the operation.

The work can be in one of these states:

- Request

  The linked production is still waiting.

- Draft

  The production has started but no cycle was already consumed.

- Waiting

  There are some draft cycles planned.

- Running

  There is at least one running cycle.

- Finished

  All the cycles are done (or cancelled).

- Done

  The production is done.

The works are created on the waiting production using the linked routing. For each step of the routing, a work is created with the operation. The work center is set if the operation has a work center category, by choosing a children work center of this category. Or if the operation has no category, it is the production work center that is used.

Cycle

Cycles are used to count the consumption and the duration of the work. It also records the effective cost from the work center.

The cycle can be in one of this states:

- Draft
- Running
- Done
- Cancelled

Production Work Timesheet Module

The production work timesheet module allows to enter timesheet for production works.

4.5.7 Project

Project

Fundamentals for project management.

Invoice

Invoices the projects.

Plan

Plans the projects.

Revenue

Follows cost and revenue of projects.

Project Module

The Project module provides the concepts of project and task and the basis for simple project management.

Work Effort

The Work Effort model is used for creating both projects and tasks. This allows for instance to transform a task into a project if it gets bigger and need to be split. The following fields are defined on the model:

- Name: The name of the Project/Task.
- Type: Can be *Project* or *Task*.
- Status: The current status of the work.
- Parent and Children: Define the tree structure of projects and tasks.
- Party and Party Address: The optional party (and the contact address) for which the project is made. Available on projects.
- Timesheet, start and end: Allow to enter timesheet for this work.
- Effort: The estimated effort of a task.
- Total Effort: Available on projects. Gives the total effort of the sub-tasks (I.E. tasks of the project and tasks of the sub-projects) of the current project.
- Progress: The progression on the task.
- Total Progress: Gives the total of progress of the sub-tasks.
- Comment: A description.

Work Status

The Work Status defines the possible status of projects and tasks. A minimal progress can be defined to enforce on works in this status.

Project Invoice Module

The Project Invoice module adds invoice methods on project. The methods are:

- Manual: Tryton doesn't create any invoice.
- On Effort: The invoices are created based on the *Effort* hours for all children works with 100% progress.
- On Progress: The invoices are create proportionally to the *Progress* of the *Effort* hours of each children work.
- On Timesheet: The invoices are created based on the timesheets encoded on all children works.

Project Plan Module

The Project Plan module adds planning features on top of the Project module.

Allocation

An allocation links together an employee, a task and a percentage. It defines which part of his time the employee should dedicate to this task.

Work

The Work model is extended to support planning features:

- Tasks dependencies: Each work may have predecessor and successors. Predecessor and successor must share the same parent project.
- tasks leveling, I.E. automatically delay some task to avoid overallocation of resources.
- Early Start and Late End computation: If Constraint Start and Constraint End are set on a work, on its parent work or on one of the predecessors/successors, the Early Start and Late End dates (but also Late Start and Early Finish dates) are computed automatically.
- Resource allocation: Each task may allocate one or more resource (I.E. a certain amount of time of an employee).

The *Task Leveling* wizard allow to level a group of tasks to avoid eventual overallocation of resources, It will delay some tasks depending on task precedencies and task sequences.

Project Revenue Module

The project revenue module computes revenue and cost per task and project. The revenue uses the list price of the product. If the product's unit of measure is time based, the revenue is computed as the product of the price and the hours of effort otherwise the price is considered as fixed. The cost is computed by summing the cost of all the linked time sheets and the linked purchase lines.

4.5.8 Purchase

Purchase

Fundamentals to follow purchases.

Amendment

Amend purchases in processing.

History

Manages revisions.

Invoice Line Standalone

Generates invoice lines.

Request

Collects the purchase requests.

Request Quotation

Asks quotation for the purchase requests.

Requisition

Requires purchase by employees.

Secondary Unit

Adds a secondary unit of measure.

Shipment Cost

Computes shipment cost.

Purchase Module

The purchase module defines the Purchase model.

Purchase

The purchase is mainly defined by a party from which the products will be purchased and a list of purchase lines, each one containing a product and a quantity. Here is the extensive list of the fields, most of them are optional or completed with sensible default values:

- Party: The supplier.
- Contact: The contact who received the order.
- Invoice Party: An optional different party that sends the invoice.
- Invoice Address: The invoice address of the supplier.
- Supplier Reference: Allow to keep track of the supplier reference for this order.
- Description: An optional description for the order.
- Number: The internal reference of the purchase (will be generated automatically on confirmation).
- Reference: The optional external reference of the order.
- Purchase Date: The date the purchase is made.
- Payment Term: Define which payment term will be use for the future invoice.
- Warehouse: Define the warehouse where the shipment will be made.
- Currency: define the currency to use for this purchase. All product prices will be computed accordingly.
- Purchase Lines:

  - Type: The type of the line. The default value is *Line* which means that the current purchase line contains the fields defined hereunder. The other values of Type (*Comment*, *Subtotal*, *Title*) are used to add extra lines that will appear on the report, thus allowing to easily customise it.
  - Sequence: Allow to order lines. The value of this field is also updated with a drag and drop between the lines.
  - Product: An optional reference to the product to purchase.
  - Description: The description of the product to purchase.
  - Quantity: The quantity to purchase.
  - Unit: The unit of measure in which is expressed the quantity.
  - Unit Price: The unit price of the product expressed in the currency of the purchase.
  - Amount: The amount of the current line (Unit Price multiplied by Quantity).
  - Delivery Date: The computed date at which the product is expected to be delivered.
  - Taxes: The list of taxes that will be applied to the current line.
- Invoice State: The state of the invoice related to the purchase.
- Shipment State: The state of the shipment related to the purchase.
- Untaxed: The untaxed amount.
- Tax: The tax amount.
- Total: The total amount.
- State: The state of the purchase. May take one of the following values: Draft, Quotation, Confirmed, Cancelled.
- Company: The company which issue the purchase order.
- Invoice Method: May take one of the following values:
  - Base on order: The invoice is created when the purchase order is confirmed.
  - Base on shipment: The invoice is created when the shipment is received and will contains the shipped quantities. If there are several shipments for the same purchase, several invoices will be created.
  - Manual: Tryton doesn't create any invoice automatically.
- Comments: A text fields to add custom comments.
- Invoices: The list of related invoices.
- Moves: The list of related stock moves.
- Shipments: The list of related shipments.
- Return Shipments: The list of the related shipment returns. If a Supplier Return location is defined on warehouse it will be used on return shipments as origin location. Otherwise the warehouse storage location will be used.

The *Purchase* report allow to print the purchase orders or to send them by mail.

Purchase Amendment Module

The purchase amendment module allows you to change purchases that are being processed and keep track of the changes. An amendment is composed of action lines which can:

- Recompute taxes (if the supplier tax rules or product taxes have changed)
- Change the payment term

- Change the party and the address
- Change the warehouse
- Change a purchase line:
  - the product (for one with the same UoM category)
  - the quantity and unit of measure
  - the unit price
  - the description

When the amendment is validated the purchase is updated and given a new revision. Generated documents (like moves and invoices) that are still in a draft state are replaced with new ones based on the new values.

Purchase History Module

The purchase history module activates the historization of the purchase and adds a revision counter which increases each time the purchase is reset to draft.

Purchase Invoice Line Standalone Module

The purchase invoice line standalone module makes purchase to generate invoice lines instead of invoices.

Purchase Request Module

The Purchase Request module introduces the concept of Purchase Requests which are central points to collect purchase requests generated by other process from Tryton.

Purchase Request

A purchase request defines the need to purchase a product. It is linked to a warehouse and possibly a supplier, it contains a ideal purchase date and an expected supply date. A purchase request can be in one of these states:

- Draft

  A draft purchase request is a request that is not linked to a purchase.

- Purchased

  A Purchased request is a request that is linked to a purchase which is not in state Done or Cancel.

- Done

  A purchase request is in state Done if it is linked to a Purchase which is in state Done.

- Cancel

  A purchase request is in state Cancel if it is linked to a Purchase which is cancelled.

A wizard allows to create purchases based on a selection of draft purchase requests. The new purchases contains one purchase line by product and unit and group them by warehouses and by suppliers. Once the purchases are created, the corresponding purchase requests are set to the state Purchased.

Purchase Request For Quotation Module

The Purchase Request for Quotation module allows users to ask quotations from selected purchase requests to different suppliers. Each request will collect quotation information from the supplier. The selection of the quotation is done by taking either the preferred_quotation field if not empty otherwise the first one ordered from the received quotations.

Quotation

- Supplier: The supplier.
- Company: The company which issue the request quotation.
- Supplier Address: The address of the supplier.
- Lines:
  - Product: An optional reference to the product to quote.
  - Description: The description of the product to quote.
  - Supply Date: The expected date to supply.
  - Quantity: The quantity to quote.
  - Unit: The unit of measure in which is expressed the quantity.
  - Unit Price: The unit price of the product expressed in the currency.
  - Currency: define the currency to use for this quotation. All product prices will be computed accordingly.
- State: The state of the quotation. May take one of the following values: *Draft*, *Sent*, *Received*, *Rejected*, *Cancelled*.

Purchase Requisition Module

The Purchase Requisition module allows users to create their purchase requisitions. Those requisitions will be approved or rejected by the Approval group. On approval, purchase requests will be created.

Requisition

- Employee: The requester.
- Description: The description of the purchase requisition.
- Supply Date: The expected date to supply.
- Currency: define the currency to use for this requisition. All product prices will be computed accordingly.
- Warehouse: Define the warehouse where the shipment will be made.
- Purchase Requisition Lines:
  - Supplier: The supplier.
  - Product: An optional reference to the product to request.
  - Description: The description of the product to request.
  - Quantity: The quantity to request.
  - Unit: The unit of measure in which is expressed the quantity.

  - Unit Price: The unit price of the product expressed in the currency of the purchase requisition.
  - Amount: The amount of the current line (Unit Price multiplied by Quantity).
- Total: The total amount.
- State: The state of the purchase requisition. May take one of the following values: Draft, Waiting, Rejected, Processing, Done, Canceled.
- Company: The company which issue the purchase requisition.

Purchase Secondary Unit Module

The purchase secondary unit module adds a secondary unit of measure on purchase lines. The secondary quantity and unit price are kept synchronized with the quantity and unit price. The secondary unit is defined on the product supplier or on the product with its factor against the purchase unit.

Read the Docs readthedocs.org

Page not available

4.5.9 Sale

Sale

Fundamentals to follow sales.

Advance Payment

Manage advance payment on sale.

Amendment

Amend sales in processing.

Complaint

Solves customer complaints.

Credit Limit

Adds sales to the credit limit.

Extra

Add extra lines.

History

Manages revisions.

Invoice Grouping

Groups invoices.

Opportunity

Tracks leads and opportunities.

Payment

Pays sales before posting the invoice.

Price List

Applies price list.

Product Customer

Adds customer's name and code for product.

Promotion

Applies promotions.

Promotion Coupon

Applies promotions with coupons.

Shipment Cost

Computes shipment cost.

Shipment Grouping

Groups shipments.

Shipment Tolerance

Define tolerance for under and over shipment.

Secondary Unit

Adds a secondary unit of measure.

Stock Quantity

Checks stock level.

Supply

Supplies from the sale.

Supply Drop Shipment

Supplies with drop shipment.

Supply Production

Supplies with production.

Sale Module

The sale module defines the Sale model.

Sale

The sale is mainly defined by a party to which the products will be sold and a list of sale lines, each one containing a product and a quantity. Here is the extensive list of the fields, most of them are optional or completed with sensible default values:

- Party: The customer.
- Contact: The contact who placed the order.
- Invoice Party: An optional different party to invoice.
- Invoice Address: The invoice address of the customer.
- Shipment Party: An optional different party where the shipment will be sent.

- Shipment Address: The address where the shipment will be sent.
- Description: An optional description for the order.
- Number: The internal reference of the sale (will be generated automatically on confirmation).
- Reference: The optional external reference of the order.
- Sale Date: The date the sale is made.
- Payment Term: Define which payment term will be use for the future invoice.
- Warehouse: Define the warehouse from which the goods will be sent.
- Currency: define the currency to use for this sale. All product prices will be computed accordingly.
- Sale Lines:
  - Type: The type of the line. The default value is *Line* which means that the current sale line contains the fields defined hereunder. The other values of Type (*Comment*, *Subtotal*, *Title*) are used to add extra lines that will appear on the report, thus allowing to easily customise it.
  - Sequence: Allow to order lines. The value of this field is also updated with a drag and drop between the lines.
  - Product: An optional reference to the product to sale.
  - Description: The description of the product to sale.
  - Quantity: The quantity to sale.
  - Unit: The unit of measure in which is expressed the quantity.
  - Unit Price: The unit price of the product expressed in the currency of the sale.
  - Amount: The amount of the current line (Unit Price multiplied by Quantity).
  - Taxes: The list of taxes that will be applied to the current line.
- Invoice State: The state of the invoice related to the sale.
- Shipment State: The state of the shipment related to the sale.
- Untaxed: The untaxed amount.
- Tax: The tax amount.
- Total: The total amount.
- State: The state of the sale. May take one of the following values: Draft, Quotation, Confirmed, Processing, Cancelled.
- Company: The company which issue the sale order.
- Invoice Method: May take one of the following values:
  - On Order Processed: The invoice is created when the sale order is processed.
  - On Shipment Sent: The invoice is created when the shipment is sent and will contains the shipped quantities. If there are several shipments for the same sale, several invoices will be created.
  - Manual: Tryton doesn't create any invoice automatically.
- Shipment Method: May take one of the following values:
  - On Order Processed: The customer shipment is created when the sale order is processed.
  - On Invoice Paid: The customer shipment is created when the invoice is paid.
  - Manual: Tryton doesn't create any shipment automatically.

One should note that choosing *On Shipment Sent* for the Invoice Method and *On Invoice Paid* for the Shipment Method is an invalid combination.

- Comments: A text fields to add custom comments.
- Invoices: The list of related invoices.
- Moves: The list of related stock moves.
- Shipments: The list of related shipments.
- Return Shipments: The list of related shipments return.

The *Sale* report allow to print the sale orders or to send them by mail.

Sale Advance Payment Module

The sale_advance_payment module adds support for advance payment management on the sale.

The module adds a new document: the Advance Payment Term that defines how the shipment or the supply processes should be managed. Either process can be completely delayed until the advance payments are paid.

Two new fields are added to *Sale*:

- *Advance Payment Term*
- *Advance Payment Conditions*

The conditions are computed at the quotation of the sale if a payment term is set.

When the sale is processed the advance payment invoices are created, final invoices will be generated when this advance payment invoices are paid. Those invoices will take into the account the amount previously paid by the customer.

Warning: If an advance payment invoice is cancelled and not recreated when processing the exception. The condition of the cancelled invoice will be concidered as met.

Advance Payment Term

It defines how advance payment invoices should be created and what they block until they are paid.

- Name: The name of the term.
- Lines:
  - Description: The description used for the invoice line.
  - Account: The account used for the invoice line. If it has default taxes, the taxes will be applied.
  - Block Supply: If checked, it prevents the creation of any supply request before the payment of the advance payment invoice.
  - Block Shipping: If checked, the shipments can not be packed before the payment of the advance payment invoice.
  - Invoice Delay: The delay to apply on the sale date for the date of the advance payment invoice.
  - Formula: It computes the amount of the invoice line.

Sale Amendment Module

The sale amendment module allows you to change sales that are being processed and keep track of the changes. An amendment is composed of action lines which can:

- Recompute taxes (if the customer tax rules or product taxes have changed).
- Change the payment term
- Change parties and addresses
- Change the warehouse
- Change a sale line:
  - the product (for one with the same UoM category)
  - the quantity and unit of measure
  - the unit price
  - the description

When the amendment is validated the sale is updated and given a new revision. Generated documents (like shipments and invoices) that are still in a draft state are replaced with new ones based on the new values.

Sale Complaint Module

The sale_complaint module defines Complaint model.

Complaint

The complaint is mainly defined by a customer which complains about a sale or an invoice. Actions can be taken to solve the complaint. Here is the extensive list of the fields, most of them are optional:

- Customer: The customer.
- Address: The main address of the customer.
- Date: The date the complaint is filled.
- Number: The internal reference of the complaint (will be generated automatically on creation).
- Reference: The optional external reference of the complaint.
- Employee: The employee responsible of the complaint.
- Type: The type of complaint
- Origin: The original document for which the complaint if filled.
- Company: The company against which the complaint is filled.
- Description: The description of the complaint.
- Actions: The actions to take to solve it.
- State:
  - Draft
  - Waiting: The complaint is waiting for approval.
  - Approved: The complaint has been approved by a sale admin.

  - Rejected: The complaint has been rejected by a sale admin.
  - Done: The complaint's actions have been executed.
  - Cancelled

Action

A complaint action defines an action to execute to solve the complaint. There are two types of action: *Create Sale Return* and *Create Credit Note*. When the origin of the complaint is a line, only this line will proceeded and it will be possible to define the quantity and the unit price otherwise it is the all document.

Type

It defines the type of complaint per document: *Sale*, *Sale Line*, *Customer Invoice* and *Customer Invoice Line*.

Sale Credit Limit

The sale_credit_limit module adds confirmed sale but not yet invoiced to the "Credit Amount" of the Party and check the credit limit of the party when confirming a sale.

Sale Extra

The sale_extra module allows to add extra line on sale based on criteria.

The extra products are added when the sale goes into quotation but the added lines can be modified when going back to draft.

The criteria are defined by the *Sale Extras* of the *Price List*.

Sale Extra

Each matching *Sale Extra* is evaluated to add an extra line. The criteria are the fields:

- *Price List*
- *Start/End Date*
- *Sale Amount*: If untaxed sale amount is greater or equal (in the price list company currency).

Sale Extra Line

Once a *Sale Extra* is selected, its first line that match the line's criteria is used to setup the extra line. The criteria are the fields:

- *Sale Amount*: If the untaxed sale amount is greater or equal (in the price list company currency).

The sale line is setup using the fields:

- *Product*: The product to use on the sale line.
- *Quantity*: The quantity to set on the sale line (in the *Unit*).
- *Unit*: The unit of measure to use.

- *Free*: Set unit price of the sale line to zero, otherwise to the sale price.

Sale History Module

The sale history module activates the historization of the sale and adds a revision counter which increases each time the sale is reset to draft.

Sale Invoice Grouping Module

The `sale_invoice_grouping` module adds an option to define how invoice lines generated from sales will be grouped.

A field is added to the *Party*:

- *Sale Invoice Grouping Method*: The method used when grouping invoice lines.

If the Standard method is used, invoice lines generated will be added to the first matching invoice found. If no invoice matches sale attributes then a new one will be created. Invoices not created by the sale process are not taken into account when looking for a candidate invoice to add invoice lines to.

Sale Opportunity Module

The sale_opportunity module defines the lead/opportunity model.

Lead/Opportunity

The lead and opportunity are defined by the same record but with different state. Depending on the state, fields of the record become mandatory. Here is the extensive list of the fields, most of them are optional:

- Party: The customer.
- Contact: The contact who placed the order.
- Address: The main address of the customer.
- Description: The description of the lead/opportunity.
- Number: The internal reference of the lead/opportunity (will be generated automatically at creation).
- Reference: The optional external reference of the order.
- Amount: The estimated revenue amount.
- Currency: Define the currency of the amount.
- Probability: The probability of conversion.
- Company: The company which issue the lead/opportunity.
- Employee: The employee responsible of the lead/opportunity.
- Start Date: When the lead started.
- End Date: When the lead ends (automatically set on win, cancel or lost).
- Payment Term: Define which payment term will be used for the future invoice.
- Comment: A text field to add custom comments.
- State: The state of the lead/opportunity. May take one of the following values:

  - Lead
  - Opportunity
  - Converted
  - Won
  - Cancelled
  - Lost
- Lines: A list of *Lead/Opportunity* line
- Sales: The list of sales converted from this opportunity.
- History: The list of changes made over time to the lead/opportunity.

The first time an opportunity is converted, a sale is created base on the information of the opportunity and the lines. The amount of the opportunity is later updated with the sales amount. Once all the sales of an opportunity are confirmed (or cancelled with at least one confirmed), the opportunity is won. If all the sales of an opportunity are cancelled than the opportunity is lost.

Lead/Opportunity Line

A lead/opportunity line define a quantity of product that are expected to be sold.

Reports

The sale_opportunity module defines also some reports:

- Opportunities per Employee.
- Opportunities per Month.
- Opportunities per Employee per Month.

which all show:

- The number of leads/opportunities.
- The number of converted opportunities.
- The convertion rate.
- The number of won opportunities.
- The winning rate.
- The number of lost opportunities.
- The total amount of leads/opportunities.
- The total amount of converted opportunities.
- The convertion amount rate.
- The total amount of won opportunities.
- The winning amount rate.

Sale Payment Module

The sale_payment module extends *Sale* to allow payments prior to the creation of any invoice.

A field *Payments* is added on the sale which can be filled in quotation state. The sale can not be reset to draft or cancelled if there are no failed payment linked.

The payment authorization of the full sale amount is used as confirmation of the sale.

When an invoice from a sale is posted, its payments are used to pay invoice's *Lines to Pay*.

Sale Price List Module

The sale price list module adds support for price list on sale. A price list can be set per party or as default.

Sale Product Customer Module

The sale_product_customer module defines customer's names and codes for products and/or variants.

A reference to the customer's product is added to the sale line.

Product Customer

- Party: The customer.
- Name: The customer's product name.
- Code: The customer's product code.

Sale Promotion

The sale_promotion module allows to apply promotions on sale based on criteria.

The promotion is applied by changing the unit price of the affected lines when the sale goes into quotation but the unit price is restored when going back to draft.

Sale Promotion

Each matching *Sale Promotion* is considered for application but only those who reduces the unit price of the lines are really applied. The criteria are the fields:

- *Company*: The company should be the same as the sale.
- *Price List*: The sale price list.
- *Start Date*/*End Date*: The period for which the promotion is valid.
- *Quantity*: The sum quantity of the sale lines which have the same *Unit*.
- *Products*: The list of products to apply the promotion.
- *Categories*: The list of product categories to apply the promotion.

The new unit price is computed by the field *Formula*.

Sale Promotion Coupon Module

The sale_promotion_coupon module adds coupon to the promotions.

A promotion with coupons will be applied only if a linked coupon has been added on the sale.

Promotion Coupon

It defines the behaviours of a sets of coupon numbers for a promotion:

- *Name*: The name of the coupon.
- *Promotion*: The promotion to which the coupon is attached.
- *Number of Use*: How much time a coupon number can be used.
- *Per Party*: How the usage is counted.
- *Numbers*: The list of numbers which must be unique.

Sale Shipment Cost Module

The sale_shipment_cost module adds shipment cost for sale.

Two new fields are added to *Sale* and *Sale Configuration*:

- *Carrier*: The carrier used for the sale.
- *Shipment Cost Method*: The method used to compute the cost.
  - *On Order*: The cost will be computed once for the sale.
  - *On Shipment*: The cost will be computed for each shipments.

At the quotation if a carrier is selected a new line is appended with the shipment cost but the added line can be modified when going back to draft.

Three new fields are added to *Customer Shipment*:

- *Carrier*: The carrier used for the shipment.
- *Cost*: The cost of the shipment.
- *Cost Currency*: The currency of the cost.

Sale Shipment Grouping Module

The `sale_shipment_grouping` module adds an option to define how stock moves generated from sales will be grouped.

A field is added to the *Party*:

- *Sale Shipment Grouping Method*: The method used when grouping stock moves.

If the Standard method is used, stock moves generated will be added to the first matching shipment found. If no shipment matches sale attributes then a new one will be created.

Sale Shipment Tolerance Module

The `sale_shipment_tolerance` modules adds under and over shipment tolerance on the sale. If the quantity of a sale line is under shipped but inside the tolerance percentage, then the line will be considered as fully shipped and no back-order will be created. If the quantity of a sale line is over shipped more than the tolerance percentage, then a warning is raised.

Sale Secondary Unit Module

The sale secondary unit module adds a secondary unit of measure on sale lines. The secondary quantity and unit price are kept synchronized with the quantity and unit price. The secondary unit is defined on the product with its factor against the sale unit.

If the sale product customer module is activated, a specific secondary unit may be defined on the product customer. And it overrides the definition on the product.

Sale Stock Quantity

The sale_stock_quantity module check the stock quantity of the products when quoting a sale. The check will warn the user if the forecast quantity at the sale date (and further dates until next supply) is lower than the quantity sold by considering other sales and the stock forecasts.

Sale Supply Module

The Sale Supply module adds a "supply on sale option" to purchasable products. If checked, it will generate a purchase request for each sale line of this product regardless of the stock levels. Once the purchased products are received they are assigned on the customer shipments. If the purchase is cancelled the sale goes back to the default supply method.

> **Warning:** If the shiment method is *On Invoice Paid*, the purchase request will be created only when all the invoice lines are paid

Sale Supply Drop Shipment Model

The Sale Supply Drop Shipment module adds a drop shipment option on product supplier if "supply on request" is checked. When checked, the purchase request and the linked purchase have the address of customer as Delivery Address; at the confirmation of the purchase a drop shipment is created and linked to both the purchase and the sale.

Drop Shipment

A drop shipment is used when products are sent directly from the supplier to the customer without going through the warehouse. It is mainly composed of a supplier, a customer and a list of moves.

The drop shipment can be in one of this states:

- Draft

  All moves are in draft.

- Waiting

  All moves are in draft, the synchronization between the moves of the supplier and the moves to the customer has occurred.

- Shipped

  All moves from the supplier are done.

- Done

  All moves are in state Done.

- Cancel

  All moves are cancelled.

Sale Supply Production Module

The Sale Supply Production module adds a "supply on sale" option to producible products. If checked, it will generate a production request for each sale line of this product regardless of the stock levels. Once the products are produced they are assigned to the customer shipments. If the production request is cancelled, the sale goes back to the default supply method.

> **Warning:** If the shiment method is *On Invoice Paid*, the production request will be created only when all the invoice lines are paid.

4.5.10 Sale Subscription

Subscription

Fundamentals to manage subscriptions.

Asset

Add asset to subscribed service.

Sale Subscription Module

The sale subscription module defines subscription, services and recurrence rule models.

Subscription

A subscription defines how some recurring services are periodically invoiced. The invoice is generated based on the consumption of each services. Here is the extensive list of the fields:

- Party: The customer.
- Contact: The contact who placed the order.
- Invoice Party: An optional different party to invoice.
- Invoice Address: The invoice address of the customer.
- Number: The internal reference of the subscription.
- Reference: The optional external reference of the subscription.
- Description: An optional description for the subscription.
- Start Date: The date at which the subscription starts.
- End Date: The optional end date of the subscription.

- Invoice Start Date: The date at which the invoice starts to be generated.
- Invoice Recurrence: The definition of the invoicing recurrence.
- Payment Term: The payment term to apply on the invoices.
- Currency: Define the currency to use for the subscription. All service prices will be computed accordingly.
- Lines:
  - Service: A required reference to the service subscribed.
  - Description: The description of the service subscribed.
  - Consumption Recurrence: The optional recurrence of consumption of the service.
  - Quantity: The quantity consumed on each occurrence.
  - Unit: The unit of measure of the quantity.
  - Unit Price: The unit price of the service expressed in the currency of the subscription.
  - Start Date: An optional later start date than the subscription.
  - End Date: An optional earlier end date than the subscription.
- States: The state of the subscription. May take one of the following values: Draft, Quotation, Running, Closed, Canceled.
- Company: The company which issue the sale order.

A running subscription can be modified by going back to draft and edit. Some field may not more be editable if the consumption has already started. The draft subscription is momentary stopped until it is set back to running.

The consumptions are created by schedulers or by a wizard. Idem for the creation of the invoices.

Service

A subscription service defines the default consumption of a product. It is composed of the fields:

- Product: A product of type service.
- Consumption Recurrence: The recurrence at which the service is consumed.
- Consumption Delay: A delay to apply between the date the consumption is created and the date of the consumption.

Recurrence Rule

It defines combination of rules which compute the occurrence dates.

- Name: The name of the rule.
- Rules:
  - Exclusive: Define if the rule excludes the resulted dates.
  - Frequency: Daily, Weekly, Monthly, Yearly.
  - Interval: The interval of the frequency
  - By Week Day: Defines the list of weekdays where the recurrence will be applied.
  - By Month Day: Defines the list of month days to apply the recurrence to.

  - By Year Day: Defines the list of year days to apply the recurrence to.
  - By Week Number: Defines the list of week numbers (ISO8601) to apply the recurrence to.
  - By Month: Defines the list of months to apply the recurrence to.
  - By Position: Defines the list of occurrence positions.
  - Week Start Day.

The computation of occurrences is base on the python-dateutil library.

Sale Subscription Asset Module

The sale subscription asset module adds the notion of asset to the sale subscription module.

Two fields are added to the *Service*:

- *Lots*: All the lots that can be used when providing this service
- *Available Lots*: A subset of the preceding field displaying the available lots

On the *Subscription Line*, it is possible to specify the lot to use. When the subscription will be running this field will become required for service with lots.

On the *Lot*, the field *Subscribed* will point to the line of the subscription currently using this lot.

4.5.11 Stock

Stock

Fundamentals to manage warehouses.

Consignemt

Manages consignment stock from supplier or at customer warehouse.

Forecast

Forecast the stock levels.

Inventory Location

Creates inventories for many locations.

Location Move

Makes location movable.

Location Sequence

Orders locations.

Lot

Track products with lot.

Lot Shelf Life Expiration Date

Manages expiration dates.

Lot Unit

Defines unit and quantity per lot.

Package

Store packaging on shipments.

Package Shipping

Fundamentals to interact with shipping services.

Package Shipping DPD

Connects with DPD shipping service.

Package Shipping UPS

Connects with UPS shipping service.

Product Location

Defines preferred locations for products.

Secondary Unit

Adds a secondary unit of measure.

Shipment Measurements

Adds measurements to shipments.

Split

Splits moves and shipments.

Supply

Supplies warehouses.

Supply Forecast

Uses forecast to supply warehouses.

Supply Day

Compute supply date per week day.

Supply Production

Supplies warehouses with production orders.

Stock Module

The stock module defines fundamentals for all stock management situations: Locations where products are stored, moves between these locations, shipments for product arrivals and departures and inventory to control and update stock levels.

Location

Locations are generic places where products are physically or virtually stored. The following location types are defined:

- Storage

  Storage locations define real places where products are stored.

- Warehouse

  Warehouses are meta-locations which define input, storage, picking, output and lost and found locations. These locations are all of type Storage. Input and Output are locations where incoming an outgoing product are

temporally stored awaiting transportation. The storage location is often the biggest location where products are stored for middle or long periods of time. The picking location is optionally where the products are picked by the customer shipment otherwise the storage location is used. The lost and found location is the location used by inventories when correcting stock levels.

- Customer

  Customer locations are virtual locations accumulating products that have been sent to customers.

- Supplier

  Supplier locations are virtual locations accumulating products that have been received from suppliers.

- Lost And Found

  Lost And Found locations collects inventory gaps. See :ref:inventory for details.

- Drop

  Drop locations are virtual locations used as intermediary locations in the process of drop shipping.

- Production

  Production locations are used during the production of products.

- View

  View locations are virtual locations that can be used to logically group other location types.

Locations are organised in tree structures, allowing to define fine grained structures. It is possible to restrict a location to have only one level of children, this allows to improve the performance of the stock quantity computation.

Location Lead Time

It allows to define the time needed for an *Internal Shipment* between two warehouses.

Move

A move is a movement of a product in a given quantity between two locations. It may eventually defines a unit price and a currency for the products that are moved from or to another company, allowing to compute stock value at any time (and to update the cost prices if the chosen cost price method is *Average*). A move also defines a planned date (when one plan to do the move) and an effective date (when the move is actually made). Products that are used in stock move must of of type *Goods* or *Assets*. Stock levels are ignored for consumable, this means that they can be always assigned. *Service* products are ignored by the stock module.

A move can be in one of this states:

- Draft

  The initial state, used when the move is created and to define future stock movement that are planned, but still subject to modifications.

- Assigned

  An assigned move allow to reserve some products. Thus preventing other user to assign them.

- Done

  The move is in state Done when the real movement is made.

- Cancel

  A cancelled move will be ignored by the system. Only Draft or Assigned move can be cancelled. To revert a move in state Done, an opposite move must be created.

- Staging

  A phantom state used to create in advance move that should not be taken for stock computation.

A cron task runs every day and recomputes the cost price of moves if a past unit price has changed.

Product Quantities

Product quantities on each location are the sum of all moves coming from or going to this location. For quantities that are computed for a date in the past, only confirmed moves (i.e. in state Done) with an effective date inferior to the considered date are taken into account, reflecting the real situation. For future quantities, Draft and Assigned move with a planned date greater than today and smaller than the given date are also summed.

Shipment

A Shipment define a group of moves happening at the same date and around the same location.

Supplier Shipment

A supplier shipment is used when products are received from a supplier. It is mainly composed of a party (the supplier), a location (the warehouse in which the products are coming) and two list of moves:

- Incoming moves

  The moves between the supplier location and the input location (as defined on the warehouse).

- Inventory moves

  The inventory moves are between the input location and the storage location (or one of his child locations).

If the storage location is configured as the same as the input location only incoming moves are created.

The supplier shipment can be in one of this states:

- Draft

  Incoming moves and inventory moves (if they exist) are in draft.

- Received

  Incoming move are set in state Done, inventory moves are created if necessary.

- Done

  Inventory and incoming moves are in state Done.

- Cancel

  All moves are cancelled.

Customer Shipment

A customer shipment is used for sending products to customer. It is mainly composed of a party (the customer), a location (the warehouse out of which the product are going) and two list of moves:

- Inventory moves

  The moves between the picking or storage location and the output location of the warehouse

- Outgoing moves

  The moves between the output location of the warehouse and a customer location.

If the picking or storage location is configured as the same as the output location, then only outgoing moves are created and no assignation is done.

The customer shipment can be in one of this states:

- Draft

  Outgoing moves and inventory moves (if they exist) are in draft.

- Waiting

  When a customer shipment is set to waiting, the inventory moves are created (or completed) to balance the outgoing moves. The waiting state also means that the shipment should be processed.

- Assigned

  The assigned state is when products have been assigned (or reserved) from the storage locations.

- Packed

  The packed state is when the inventory moves have been made, i.e when the products have been physically moved to the outgoing locations.

- Done

  The shipment is Done when the outgoing moves have been made, e.g. when a truck left the warehouse.

- Cancel

  A shipment which is not yet completed (not in state Done) can be cancelled at any time. This also cancel all the moves.

Internal Shipment

An internal shipment is used for sending products across locations inside the company. It is mainly composed of two locations and a list of moves. It can be in one of these states:

- Draft

  The moves (if they exist) are in draft.

- Waiting

  The waiting state means that the shipment should be processed.

- Assigned

  The assigned state is when products have been assigned.

- Done

  The shipment is Done when the moves have been made.

- Cancel

  A shipment which is not yet completed (not in state Done) can be cancelled at any time. This also cancel all the moves.

Inventory

Inventories allow to control and update stock levels. They are mainly composed of the inventoried storage location and a list of inventory lines. Inventory lines consist of a product and it's default unit of measure, an expected quantity and the real quantity (the real products on the shelves).

A button allows to auto-complete inventory lines with respect to the expected quantities for each product in the location. Another button allows to launch a wizard to count products by adding the quantity to the existing matching line.

When the inventory is confirmed, moves are created to balance expected quantities and real ones.

Warning: Inventories must not be used to import initial stock levels. Individual moves from supplier to each locations must be used with the cost price as unit price.

Product

The cost price of a product can only be modified using the "Modify Cost Price" wizard once it is associated with stock moves. The wizard stores, for each template or product the cost price revision. This revision contains a formula that compute the new cost price based on the current one. E.g. *cost_price * 0.9* to reduce the cost price by 10%. The cost price revisions are applied at the beginning of the stored date when the cost price of a product is re-computed.

Warning: If the user modifies a revision manually, they must also run the "Recompute Cost Price" wizard.

Stock Consignment Module

The stock consignment modules allow to manage consignment stock from supplier or at customer warehouse.

The consignment stock from supplier is managed by creating a supplier location under the company's warehouse storage. The location can be filled using an Internal Shipment from the external supplier location. The products are used also by using an Internal Shipment from the consignment location to a storage location. In this case, a supplier invoice line is created for the supplier defined on the location.

The consignment stock at customer warehouse is managed by creating a storage location under the customer location. The location can be filled using an Internal Shipment from a warehouse. It is possible to define a lead time between the warehouse and the storage location. The products are used also by using an Internal Shipment from the consignment location to a customer location. In this case, a customer invoice line is created for the customer defined on the location.

It is allowed to make inventory for those consignment locations.

A new field is added to Location:

- Consignment Party: The party invoiced when consignment is used.

Stock Forecast Module

The stock forecast module provide a simple way to create stock moves toward customers with a date in the future. This allow other stock mecanism to anticipate customer demand.

Forecast

The forecast form contains:

- A location from which the products will leave.
- A destination (which is a customer location).
- Two dates defining a period in the future.
- A company
- A list of forcast lines with:
  - A product
  - A quantity which represent the total demand for the period
  - A minimal quantity for each move.
  - A unit of measure.

The "Complete Forecast" button allow to auto-complete forecast lines based on previous stock output for dates in the past.

The forecasts are deactivated automatically when their period has passed.

Forecast States

Draft

It is the initial state and the state used for edition. No moves are linked to the forecast lines

Done

Once in state done, moves are created for each forecast line:

- They are spread homogeneously between the two dates of the forecast.
- Move quantities are bigger or equal to the minimal quantity set on the forecast line.

Cancel

On a cancelled forecast all existing moves are cancelled and the form is readonly.

Stock Inventory Location Module

The Stock Inventory Location Module add a new wizard *Create Inventories* under the *Inventories* sub-menu.

This wizard first asks the user to select the locations, the products and the categories of product that he wants to create inventories for and opens them in a new tab.

Stock Location Move Module

The stock location move module allows to define some *Locations* as movable (like palette).

Such locations can be moved using an *Internal Shipment*. The parent location is changed for the destination location when the shipment is done. If there is a transit location, the locations will be first moved to this one when shipped. A reservation mechanism prevent to assign the same locations at the same time.

If a *Customer Shipment* or a *Supplier Return Shipment* empties a movable location, it will automatically deactivate it.

Stock Location Sequence Module

The stock location sequence module adds ordering to location.

Stock Lot Module

The stock lot module defines lot of products.

Lot

A lot is by default just a number to tag moves. It is possible to define per product when the lot is required by selecting the type of location involved in the move.

Stock Lot Shelf Life Expiration Date

The stock_lot_sled module adds two fields on lot of products:

- Shelf Live Expiration Date
- Expiration Date

And it defines on the product the default time for those fields and on the stock configuration the Shelf Live Delay.

When the shelf life of a lot expires in less than the configured shelf life delay, it is no more used to compute the forecast quantity of the stock.

Stock Lot Unit Module

The *stock_lot_unit* module allows to define a unit and quantity on stock lot.

Lots with unit have the following properties:

- no shipment may contain a summed quantity for a lot greater than the quantity of the lot.
- no move related to a lot with a unit may concern a quantity greater than the quantity of the lot.

The *Lot Unit* field is added to the product. This defines the unit to set on new lot.

Stock Package Module

The stock package module allows to store packaging information about customer and supplier return shipments.

Package

A package groups together some moves from the same shipment. The packages can be organized in a tree structure. If a package is defined on a shipment, this one could be set to *Done* only if all moves are packaged.

Stock Package Shipping Module

This module is the base module required to interact with shipping service providers.

Carrier

The Carrier model adds the following field:

- *Shipping Service*: The shipping service of the carrier.

This field is programmatically filled by the modules providing support for shipping companies.

Package Type

The Package Type model has been added the following fields:

- *Length*: The length of the packages of this type
- *Length Unit*: The unit of measure of this length
- *Length Digits*: The precision of length
- *Height*: The height of the packages of this type
- *Height Unit*: The unit of measure of this height
- *Height Digits*: The precision of height
- *Width*: The width of the packages of this type
- *Width Unit*: The unit of measure of this width
- *Width Digits*: The precision of width

Package

The Package model has been added the following fields:

- *Shipping Reference*: The shipping reference provided by the shipping service
- *Shipping Label*: The shipping label provided by the shipping service
- *Weight*: A function field computing the weight of the package with its content

Shipment Out

The Shipment Out model will check once in the Packed state if the shipment is a valid shipment for the shipping service. He does that by calling a method that is by convention named `validate_packing_<shipping service>`.

Once a shipment is packed, the user can create the shipping for each packages with the shipping service by clicking on the *Create Shipping* button. This button triggers a wizard that is overridden in shipping service specific modules. The starting state of the wizard is a `StateTransition`. Its linked method is overridden in shipping service modules in order to communicate with the service.

Stock Package Shipping DPD Module

The Stock Package Shipping DPD module allows you to generate the DPD label using the DPD webservices. DPD has many different web services, the module supports:

- *public-ws.dpd.com*

Carrier Credential

This model allows to define the credential used to connect the the DPD API. The credentials will be automatically retrieved based on the company of the Shipment.

- *Company*: The company for which those credentials are valid
- *User ID*: The User ID provided by DPD.
- *Password*: The Password used to access the DPD API.
- *Server*: Are those credentials used for accessing the Testing or the Production server.

Stock Package Shipping UPS Module

The Stock Package Shipping UPS module allows you to generate the UPS labels per package using the UPS webservices.

The credential for the webservice is specified by the following fields:

UPS Carrier Credential

This model allows to define the credential used to connect the UPS API. The credentials used to connect to the UPS API will be automatically retrieved based on the company of the Shipment.

- *Company*: The company for which those credentials are valid
- *User ID*: The User ID provided by UPS.
- *Password*: The Password used to access the MyUPS Portal.
- *Account Number*: The account number UPS assigned to you when you requested the API credentials (also known as the Shipper Number).
- *License*: The License number UPS assigned to you when you requested the API credentials
- *Server*: Are those credentials used for accessing the Testing or the Production server of UPS.
- *Use Metric*: Use Metric units when communicating with the UPS webservice.

Carrier

The Carrier model is extended with the following fields:

- *Service Type*: The UPS service type requested
- *Label Image Format*: The Image Format used for the label sent by UPS
- *Label Image Height*: The Height of the label sent by UPS.

Package Type

The Package Type model is extended with the following field:

- *UPS Code*: The UPS Code of the package.

Stock Product Location Module

The Stock Product Location Module adds on the product form a list of preferred location by warehouse. This list is used when a supplier shipment is received: the auto-generated Inventory Moves will use as default destination the preferred locations associated to the current warehouse.

Stock Secondary Unit Module

The stock secondary unit module adds a secondary unit of measure on the stock move.

Stock Shipment Measurements Module

The Stock Shipment Measurements module adds weight and volume on shipments and packages. They are computed using the measurement and the quantity of their moves.

Stock Split Module

The Stock Split module adds on the stock move a wizard that allows to split them. The move is split into moves of *Quantity*. If *Counts* is set, it will be split only this number of times. On occasion there can be a move with the remaining quantity.

It also adds on *Customer*, *Supplier Return* and *Internal* shipments a wizard that allows to split them. The selected moves of the shipment are shifted to a new copy of the shipment.

Stock Supply Module

The Stock Supply module add automatic supply mechanisms and introduce the concepts of order point.

Order Point

An order point define minimum, maximum and target quantities for a product on a location.

- The minimum quantity is the threshold quantity below which the provisioning process will be triggered.
- The maximum quantity is the threshold quantity above which the overflowing process will be triggered.

- The target quantity is the quantity that will be found in the location after the provisioning / overflowing process has been completed.

An order point also define a type which can be:

- Internal

  An Internal order point is defined on a Storage location, it also defines a provisioning and/or an overflowing location. If the minimum quantity is reached it will result in the creation of an internal shipment between the provisioning location and the Storage location. If the maximum quantity is reached it will result in the creation of an internal shipment between the storage location and the overflowing location.

- Purchase

  A Purchase order point is defined on a warehouse location. If the minimal quantity is reached on the warehouse it will result in a purchase request.

The internal shipments and purchase requests are created by the supply wizard with respect to stock levels and existing shipments and requests. The stock levels are computed between the next two supply dates computed over the Supply Period from the configuration (default: 1 day). If the stock level of a product without order point on the given warehouse is below zero, a purchase request is also created. The same happens if the stock level of a storage location with a provisioning location is below zero. Likewise, if the stock level of a storage is above zero and an overflow location is defined on the location then an internal shipment will be created.

Stock Supply Forecast Module

The stock supply forecast module takes forecast into account to compute purchase requests.

Stock Supply Day Module

The Stock Supply Day module adds a Week Days list on the Product Supplier form. This allow to restrict the supply week days for each supplier on each product. If no days are defined for a supplier a supplying may happens at any day of the week.

For example, those week days are taken into account when computing the planned date of stock moves created by the purchase (defined in the Purchase module) and by the scheduler that generates the Purchase Requests (defined in the Stock Supply module).

Stock Supply Production Module

The Stock Supply Production module adds automatic supply mechanisms via production request.

Production Request

A production request is a production in the state request. It defines the need to produce a product.

Order Point

It adds a new type of Order Point:

- Production

  A Production order point is defined on a warehouse location. If the minimal quantity is reached at any time on the warehouse it will result in a production request.

The production requests are created by the supply wizard with respect to stock levels and existing requests. The stock levels are computed on the supply period define in the production configuration.

4.5.12 Timesheet

Timesheet

Fundamentals to track time spend.

Cost

Tracks employee cost.

Timesheet Module

The timesheet module allow to track the time spent by employees on various works. This module also comes with several reports that show the time spent by employees on works following various time periods.

Work

A work is a generic concept that encompass all activities from simple tasks to long-running projects. It can be alone or linked to another document. The Work model contains the following fields:

- Name or Origin: The name or the origin of the work.
- Active: A checkbox that allow to disable the work.
- Company: The company for which the work is (or was) executed.

Timesheet Line

A timesheet line expresses the fact that one employee spends part of their time doing specific work on a given date. An optional Description field allows extra information to be recorded about what has been done.

Timesheet Cost Module

The timesheet cost module adds cost price per employee.

4.5.13 Bank Module

The bank module defines the concept of bank and account.

Bank

A bank links a party with a BIC.

Bank Account

A bank account is defined by a *Bank* and at least one number.

- The *Bank* is the bank where the account is set.
- The *Owners* are the parties who own the account.
- The *Currency* is the default currency of the account.
- The *Numbers* are the different possible number that identifies the account. There are two types defined by default:
  - IBAN
  - Other

4.5.14 Company

Company

Defines company and employees.

Work Time

Defines work time per company.

Company Module

The company module defines the concepts of company and employee and extend the user model.

Company

The company model extends the party model and add several fields: the currency, the list of employees and header and footer texts for reports. There is also a parent company field which allow to setup companies in a tree structure. The company model represent the actual organisation the users of Tryton are members of.

Employee

The employee model extend the party model with a company field. The employee model represent the actual employees of the companies defined in Tryton. An employee can be optionally linked to a user trough the user form.

User

Are added to the user model: a main company, a company and an employee field. The company field defines the current company of the user, this current company will influence the data this user see in Tryton: most of the records that are linked to a company will only be available for users in this very company. The main company define which current company a user can choose: either the main company itself or one of the children companies.

Company Work Time Module

The Company Work Time module adds 4 new fields (Hours per Work Day, Hours per Work Week, Hours per Work Month, Hours per Work Year) on the company form that allow to define how much hours are spent by an employee in a day, a week, a month and a year of work.

4.5.15 Carrier

Carrier

Fundamentals to manage carriers.

Percentage

Computes cost based on percentage.

Weight

Computes cost based on weight.

Carrier Module

The carrier module defines the concept of carrier.

Carrier

A carrier links a party with a product and cost method.

- The *Product* is the carrier service.
- The *Carrier Cost Method* defines how to compute the carrier cost. By default there is only the *Product Price* method which takes the *List Price* of the *Product* as sale price and the *Cost Price* of the *Product* as purchase price.

Carrier Selection

A carrier selection defines the country of origin and destination where a carrier can deliver the products.

- The *Sequence* is used to order the Carrier Selections.
- *Active* allows to disable a Carrier Selection.
- *From Country* defines the Country of origin for this Carrier Selection. Empty value act as a wildcard.
- *To Country* defins the Country of destination for this Carrier Selection. Empty value act as a wildcard.

Carrier Percentage Module

The carrier percentage module adds a cost method based on percentage.

Carrier Weight Module

The carrier weight module adds a cost method based on weight. The price is computed by finding the line for which the weight is greater or equal but smaller than the next line.

4.5.16 Country Module

The country module defines the concepts of country and subdivision and comes preloaded with the ISO 3166 list of countries and subdivisions thanks to the pycountry module.

Country

A country is defined by the full name of the country, a code (the ISO country code made of two chars) and the list of subdivisions.

Subdivision

A subdivision is defined by a name, a code and a type (e.g, Capital City, Province, Emirate, etc)

Zip

A zip links a zip code of a country to a city.

Scripts

There are two scripts:

- *trytond_import_countries* to create and update countries and subdivisions from the ISO database.
- *trytond_import_zip* to create zips by country from the geonames database.

4.5.17 Currency Module

The currency module defines the concepts of currency and rate.

Currency

A currency is defined by a name, a symbol, a code, a list of rates, a rounding factor and some formatting properties: the digits to be displayed after the decimal separator, the way the numbers should be grouped, the thousands separator, the decimal separator, the positive and negative signs and their positions, the currency symbol position and if it should be separated from the number by a space.

Rate

A rate is defined by a date and a numeric value. The date gives the time from which this rate is correct. All rates are defined implicitly with respect to the same currency (the one whose rate is 1).

Scripts

There is a scripts:

- *trytond_import_currencies* to create and update currencies from the ISO database.

4.5.18 Customs Module

The customs module allows to define customs duty based on the tariff code.

Tarif Code

It stores the Harmonized System that can be set on *Product*.

- The *Code* from the HS.
- The *Country* in case of a country specific code.
- The *Start* / *End* period of the year for which the code is valid.

Duty Rate

It stores the rate of a *Tarif Code* for a country over a period.

- The *Tariff Code*.
- The *Country* for which the rate is.
- The *Type*: *Import* or *Export*
- The *Start* and *End* date of validity.
- **The *Computation Type*:**
  - *Amount*: fixed amount with currency.
  - *Quantity*: amount (in currency) per unit of measure.

4.5.19 Dashboard Module

The dashboard module allows user to configure his dashboard.

4.5.20 Google Maps Module

The Google Maps module add a new URL field on the party addresses. This link open the Google Maps page on the default browser with the map centered on the selected address.

4.5.21 Party

Party

Fundamentals to store parties.

Relationship

Defines relation between parties.

SIRET

Adds SIRET and SIREN.

Party Module

The party module defines the concepts of party, identifier, category and contact mechanism. It also comes with reports to print labels and letters and a *Check VIES* wizard.

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, identifiers, categories, contact mechanisms and a list of addresses.

Two reports are available:

- The *Labels* report creates a document with the names and addresses of all selected parties which are preformatted to be printed on labels that can be stuck on an envelope.
- The *Letter* report create a document pre-filled with the company header, the address of the recipient, a date, a greeting, an ending and the signature of the current reader.

The *Check VIES* wizard allows to check the European VAT number identifier of parties with the VIES web service.

The *Replace* wizard allows to replace duplicate record by the original and relink all the related documents.

The *Erase* wizard allows to erase all personal data of a party from the system included the historized data and the resources attached for all the parties which were replaced by this one.

Address

An address is made of a name, a street, a zip number, a city, a country, a subdivision. A sequence allow to order them. The field *Full Address* returns the formatted address included the name of the party if the context has *address_with_party* set to True, the attention name if the context has *address_attention_party* set to a party and without the country if the context key *address_from_country* is the same as the country of the address.

Address Format

It allows to define per country and language, how addresses should be formatted.

Address Subdivision Type

It allows to define for each country which types of subdivision are allowed on the address.

Contact Mechanism

A contact mechanism is made of a type, value and comment. Type can be *Phone*, *Mobile*, *Fax*, *E-Mail*, *Website*, *Skype*, *SIP*, *IRC*, *Jabber* or *Other*.

Category

A Category is just composed of a name, thus constituting tags that can be associated to parties. Categories are organised in a tree structure.

Party Relationship Module

The party relationship module allows to define different types of relations between parties.

Each relation is defined by a relation type. A reverse relation type can be defined, so when creating a relation of a type, the reverse relation will be automatically created.

It is possible to order parties by how closely related they are to a defined party. The distance is calculated based on the number of steps it takes to get from the defined party to another. By default all the different types of relationship are considered, but this can be limited by adding `relation_usages` to the context.

Configuration

The party_relationship module use the section *party_relationship* to retrieve some parameters:

- *depth*: The maximum number of steps to consider when calculating the distance between parties. The default value is *7*.

Party SIRET Module

The party siret module adds the SIREN and SIRET on party and address.

4.5.22 Product

Product

Fundamentals to manage products.

Attribute

Adds attributes to products.

Classification

Classifies products.

Classification Taxonomic

Uses Taxonomic classification.

Cost FIFO

Adds FIFO cost method.

Cost History

Historize the cost.

Measurements

Adds measurements to products.

Price List

Computes price.

Price List Dates

Sets start and end date on price lists.

Price List Parent

Adds price list cascading.

Product Module

The Product module defines the following models: Category of Unit of Measure, Unit of Measure, Product Template, Product and Product Category.

Category of Unit of Measure

A Category of Unit of Measure is simply defined by a name.

Unit of Measure

A Unit of Measure is defined by:

- Name.
- Symbol.
- UOM category.
- Rate and a Factor (the later is the inverse of the former).
- Rounding Precision and Display Digits, used to round and display quantities expressed in the given UOM.
- Active, allow to disable a UOM.

Product category

The Product Category Model is just composed of a name. Product Categories are organised in a tree structure.

Product Template and Product

The product concept in Tryton is composed of two models: Product Template and Product.

The Product Template model contains the following fields:

- Name.
- Code, a common prefix for all products.
- Type, whose value can be *Goods*, *Assets*, *Service*.
- Category.
- List Price, the default sale price expressed in the List Price UOM. product.
- List Price UOM.
- Cost Price, the cost for one unit of this product expressed in the Cost Price UOM.
- Cost Price UOM.
- Cost Price Method, which can be *Fixed* or *Average*. Defines how the cost price should be updated. *Fixed* means that the cost price stay unchanged. *Average* means that the cost price is the average cost of all the items that are in stock. It's default value can be defined on product configuration.
- Default UOM. The default UOM for this product. Used for example to express stock levels.

- Active, allow to disable a product.

The Product model extends the Product Template with two fields: Code (aka SKU) and Description. If a sequence is set on product configuration the code will be read-only and will be filled in using the sequence. It's also possible to define a list of identifiers on product. An identifier is composed by a type and a code. The following types are available:

- International Article Number (EAN)
- International Standard Audiovisual Number (ISAN)
- International Standard Book Number (ISBN)
- International Standard Identifier for Libraries (ISIL)
- International Securities Identification Number (ISIN)
- International Standard Music Number (ISMN)

Configuration

The product module uses the section *product* to retrieve some parameters:

- *price_decimal*: defines the number of decimal digits with which the unit prices are stored. The default value is *4*.
- *uom_conversion_decimal*: defines the number of decimal digits with which the conversion rates and factors of UoM are stored. The default value is *12*.

Warning: They can not be lowered once a database is created.

Product Attribute Module

The Product Attribute module defines the following models: Attribute and Attribute Set.

Attribute

An Attribute represents a property of a product, such as its colors, dimensions and so on.

Attribute Set

An Attribute Set is linked to products and allows to assign attribute values to products.

Product Classification Module

The Product Classification module defines the tools for other modules to create classifications of products. It adds a reference field *classification* to the product template.

Product Classification Taxonomic Module

The Product Classification Taxonomic module adds the taxonomic classification to the products.

The available classifications are:

Taxon

A taxon is defined by a name and a taxonomic rank. It is stored inside a tree structure.

Cultivar

A cultivar is defined by a name, a taxon and a list of cultivar groups.

Product Cost FIFO Module

The Product Cost FIFO Module add a *FIFO* option in the Cost Method field of the product form.

The cost price of product whose cost price method is *FIFO* will be updated automatically when stock moves coming from a supplier or going to a customer are made. The page FIFO and LIFO accounting on Wikipedia gives some information about this costing method.

Product Cost History Module

The Product Cost History Module adds a *Cost History* relate on the product form showing the cost price evolution of the product. The history is based on the cost price stored on the incoming stock moves for goods and assets and based on the history table for service. When a historic cost price is needed the value is taken from this history for goods and assets.

Product Measurements

The Product Measurements module adds this following measurements to Product:

- Length
- Height
- Width
- Weight

Product Price List Module

The product price list module provides formula to compute prices per product or category.

Product Price List Dates Module

The product_price_list_dates module adds *Start Date* and *End Date* conditions to the *Price List Lines*.

Product Price List Parent Module

The product_price_list_parent module adds a *Parent* to the price list and the keyword *parent_unit_price* for the formula which contains the unit price computed by the parent price list.

4.5.23 Authentication

SMS

Authentication per SMS.

LDAP

Authentication per LDAP.

Authentication SMS Module

The SMS authentication module allows to authenticate users via SMS. There are two authentication methods *sms* and *password_sms* which can be used in the *authentications* list of the *session* section in the configuration.

The *sms* method just send a code via SMS to the user. Then the user needs to transcribe the code into the login dialog.

The *password_sms* method send a code only after the user entered a valid password. This provides a two-factor authentication method.

Both methods require that the user has a *Mobile* phone number defined otherwise he can not be authenticated with those methods.

Configuration

The configuration of the module is set in the *authentication_sms* section.

function

The fully qualified name of the method to send SMS. It must take three arguments: text, to and from. This method is required to send SMS.

from

The number from which the SMS are sent.

length

The length of the generated code. Default: 6

ttl

The time to live for the generated codes in seconds. Default: 300

name

The name used in the SMS text. Default: Tryton

LDAP Authentication Module

The LDAP authentication module allows to authenticate users via a LDAP server.

The configuration of the LDAP connection is set in the *ldap_authentication* section.

To be activated, the *ldap* method must be added to the *authentications* methods list of the *session* section of the configuration.

Configuration

uri

The LDAP URL to use to connect to the server following the RFC-2255.

bind_pass

The LDAP password used to bind if needed.

uid

The UID Attribute for authentication (default is *uid*).

create_user

A boolean to create user if not in the database.

4.5.24 Notification Email Module

The notification email module allows to define email templates which will be sent to a list of recipients when a trigger is fired on a record event. Extra reports from the same record can be attached to the email.

4.5.25 User Role Module

The user_role module allows to assign roles to user instead of groups. A *Role* is defined by a set of groups. When a role is added to a user, it overrides the existing groups. A role can be added to a user for a period.

4.5.26 Web

Shortener

Shortens URL and connects them to actions.

User

Manages external users.

Web Shortener Module

The web_shortener module allows URLs to be shortened. It counts the number of times the URL is accessed and optionally triggers action.

The module defines a route */s/<shortened_id>* which will redirect the queries to the URL registered previously with *ShortenedURL.get_url*.

Models that need to be callable from a shortened URL must define the method *shortened_url_execute*. This class method will be called from the underlying queue with the record and the keywords arguments transmitted when calling *get_url*.

Shortened URL

- Shortened URL: The shortened URL
- Redirect URL: The URL the request is redirected to
- Record: The record on which *method* will be executed
- Method: The name of the method to call on *record*
- Count: The number of times this shortened URL has been triggered

Configuration

The web_shortener modules uses the parameter from the section:

- *[web]*:
  - *shortener_base*: The base URL without path for shortened URL. The default value is composed with the configuration *[web] hostname*.

Web User Module

The web_user module provides facilities to manage external user accessing from the web.

User

A user is uniquely identified by an email and he is authenticated using a hashed password. The user can be linked to a Party.

Two actions are available:

- The *Validate E-mail* which sent an e-mail to the user with a link to an URL that ensures the address exists.
- The *Reset Password* which sent an e-mail to the user with a link to an URL to set a new password.

Configuration

The web_user module uses parameters from different sections:

- *web*:
  - *reset_password_url*: the URL to reset the password to which the parameters *email* and *token* will be added.

  - *email_validation_url*: the URL for email validation to which the parameter *token* will be added.
- *email*:
  - *from*: the origin address to send emails.
- *session*:
  - *web_timeout*: defines in seconds the validity of the web session. Default: 30 days.
  - *web_timeout_reset*: in seconds the validity of the reset password token. Default: 1 day.

Bibliography

[WP-XMLRPC] http://en.wikipedia.org/wiki/Xmlrpc

[WP-WebDAV] http://en.wikipedia.org/wiki/Webdav

[WP-CSV] http://en.wikipedia.org/wiki/Comma-separated_values

[WP-ENCOD] http://en.wikipedia.org/wiki/Character_encoding

Python Module Index

t

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W