Camelot Documentation

Release

Conceptive Engineering

May 28, 2013

CONTENTS

Release default

Date May 28, 2013

CHAPTER

ONE

WHAT'S NEW

1.1 Release 13.04.13

- Uses SQLAlchemy 0.8.0
- All default models migrated from Elixir to Declarative
- Replacements for most of the Elixir functions that are compatible with Declarative
- Search splits search strings between spaces and searches for a combination of the elements
- Russian translations
- The `camelot.model.batch_job.BatchJob` is reworked to have more robust error handling, and a batch job becomes useable as a context manager
- Decouple the `camelot.core.memento.SqlMemento` from `camelot.model.memento.Memento`, so the change tracking system becomes customizable.
- List of changes can be accessed from the form view
- Support for using an existing database through SQLAlchemy reflection
- Primary key columns are not editable by default
- Documents in print preview can be edited before printing
- Import and export have configurable columns
- Add `camelot.view.action_steps.print_preview.PrintChart` action step.
- Adapt printing of charts to matplotlib 1.0
- Fix *maximum* field attribute of rating fields in editor and delegate.
- Workaround for form window hiding on Mac
- The frozen columns feature has been removed in favor of the column groups
- The embedded form has been removed in favour of `camelot.admin.object_admin.ObjectAdmin.get_compoundir`
- Unittests cover 80% of the code
- See *Migrate from Camelot 12.06.29 to 13.04.13* for documentation on how to upgrade an existing Camelot project to the latest version.

1.2 Release 12.06.29

- `camelot_manage` has been removed, since it did not contain essential functions for the development of Camelot applications.
- Port the `camelot_example` application and *Creating a Movie Database Application* to *Declarative*
- Add a toolbar to the form view, configurable through the `camelot.admin.object_admin.ObjectAdmin.get_form_t` method.
- Move the progress widget from the removed status bar to the toolbar
- Add `camelot.admin.table.ColumnGroup` in the list view.

  _static/controls/column_group.png

- See *Migrate from Camelot 11.12.30 to 12.06.29* for documentation on how to upgrade an existing Camelot project.
- Tracking of changes goes through the `camelot.admin.object_admin.ObjectAdmin`
- Cleanup of the default Camelot models :
  - they can be used independently of each other
  - Persons, Organizations, etc. have been moved to `camelot.model.party`
  - Simplification of the underlying tables
  - The default *metadata* was moved `camelot.core.sql`
- Store user changed column width in settings and *column_width* field attribute
- `camelot.admin.not_editable_admin.not_editable_admin()` has an *actions* argument
- Reworked searching for translation files
- Portuguese (Brazil) translations
- Workaround for mainwindow bug on OS X

1.3 Release 11.12.30

- Fix inclusion of stylesheets and templates in the egg

1.4 Release 11.12.29

- Import from file wizard supports importing excel files
- A number of new `ActionStep` classes that can be used in custom `Action` classes or serve as an example :
  - `camelot.view.action_steps.change_object.ChangeObjects`
  - `camelot.view.action_steps.gui.CloseView`

  - `camelot.view.action_steps.gui.MessageBox`
  - `camelot.view.action_steps.select_object.SelectObject`
- Move the repository to gitorious
- The toolbar in the one-to-many and many-to-many editor are configurable using the `ObjectAdmin.get_related_toolbar_actions()` method.
- Spanish translations
- Possibility to add a close button to a form and to customize the form close action
- Filters can have a default value
- Main menu and toolbars are configurable in the `ApplicationAdmin` through the use of actions, which allows creation of reduced main windows
- Rewrite of Camelot functions behind toolbars and menus to actions, resulting in a number of `Action` classes with sample code :
  - `camelot.admin.action.application_action.ShowHelp`
  - `camelot.admin.action.application_action.ShowAbout`
  - `camelot.admin.action.application_action.Backup`
  - `camelot.admin.action.application_action.Restore`
  - `camelot.admin.action.form_action.CloseForm`
  - `camelot.admin.action.list_action.OpenNewView`
  - `camelot.admin.action.list_action.ToPreviousRow`
  - `camelot.admin.action.list_action.ToNextRow`
  - `camelot.admin.action.list_action.ToFirstRow`
  - `camelot.admin.action.list_action.ToLastRow`
  - `camelot.admin.action.list_action.ExportSpreadsheet`
  - `camelot.admin.action.list_action.PrintPreview`
  - `camelot.admin.action.list_action.SelectAll`
  - `camelot.admin.action.list_action.ImportFromFile`
  - `camelot.admin.action.list_action.ReplaceFieldContents`
- Move to SQLAlchemy 7.x
- Undefer all fields that are going to be used in a view when querying the database
- Reduction of the lines of code with 4%

1.5 Release 11.11.16

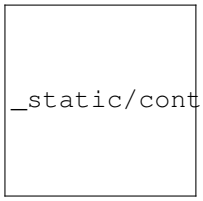
- Implementation of the new actions proposal (*Actions*), please consult the documentation and the tutorial (*Add an import wizard to an application*) of the actions to ease the migration. Most old style actions can be replaced with the new style action `camelot.admin.action.list_action.CallMethod`
- Delayed creation of widgets on tabs to improve performance for screens with lots of fields
- Move to migrate 7.1

- New splashscreen
- Italian translations
- PySide compatibility

1.6 Release 11.09.10

- Refresh reexecutes queries in the table view
- Deleted entities are grayed out in the GUI if they are deleted when visible
- Add setup.py to new projects for easy packaging
- The settings mechanism becomes plugable
- Print preview does pdf export when no printer is available
- Wizard to create a new project

- API documentation integrated with sphinx
- `camelot.core.exception.UserException`, a subclass of Exception that can be used to inform the user in a gentle way he should behave different.



- Reduced memory usage
- Experimental PySide support
- Table views are sorted by primary key to avoid row flicker
- German, French and Dutch translations
- Generation of .po files integrated with setuptools
- Fixes of VirtualAddress editor
- example renamed to camelot_example to resolve naming conflicts with other projects

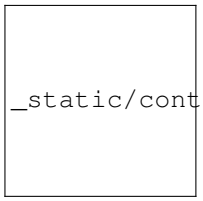
1.7 Release 11.05.13

- Faster opening of forms
- 'Home' tab with application actions
- add legend function to chart container

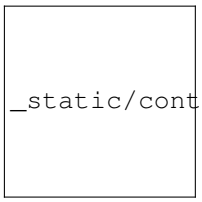
- Workspace maximizes when double clicking on tab bar
- Fix tab behavour of some editors
- Support for editing columns in the frozen part of a table view
- New DateTime Editor

  
  _static/editors/DateTimeEditor_editable.png

- More intuitive Code editor
- More intuitive navigation pane

  
  _static/controls/navigation_pane.png

- progress dialog when records are deleted
- FileEditor supports removing files after copying them
- EntityAdmin changes
  - supports objects mapped with plain SQLAlchemy, documentation on how to use this
  - confirm_delete reworked to delete_mode
  - expanded_list_search option to tune which fields show up
- ApplicationAdmin changes
  - actions_changed_signal
  - application actions show up in desktop workspace

    
    _static/controls/desktop_workspace.png

- postgres support for backup / restore
- new actions : DocxApplicationAction, PixmapFormAction
- Most editors now support background_color, editable and tooltip as dynamic attributes

1.8 Release 10.11.27

- Tab based desktop
- Faster table view

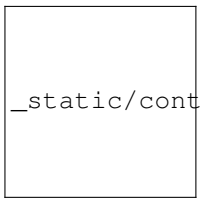
- Improved search queries
- Much more dynamic field attributes : tooltip, background_color, editable, choices, prefix, suffix, arrow
- Document merge wizard
- Support for SQLAlchemy 0.6.x
- Charts and matplotlib integration

  
  _static/editors/ChartEditor_editable.png

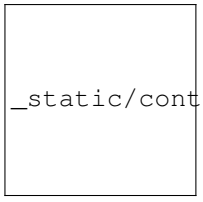
- Move from PyExcelerator to xlwt and xlrd
- Move to new style signal/slot connections
- Support for frozen columns in a table view
- Faster backup and restore

1.9 Release 10.07.02

- Expanded search and filter options

  
  _static/controls/header_widget.png

- Search works for integer, date and float fields

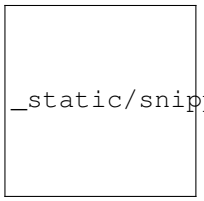
  
  _static/controls/search_control.png

- Sorting in table views and OneToMany widgets
- Importer forces validation before importing
- User translatable labels on forms
- Litebox image preview for image fields
- New editors and delegates :
  - NoteDelegate

  - LabelDelegate
  - TextBoolDelegate
- i18n improvements
- Fix date editor on windows
- Add a default model to store batch job information
- Backup and restore available from the File menu
- More documentation

1.10 Release 09.12.07

- Sqlalchemy 5.6 compatible
- Dynamic background colors and tooltips



- Generic import wizard
- The busy indicator in the status bar
- Support for lazy translations
- Remove PIL dependency and only depend on QImage
- Support multiple levels of class inheritance in the model
- Various bugfixes, usability and speed improvements
- Code cleanup

Contents:

CHAPTER

TWO

TUTORIALS

This section contains various tutorials that will help the reader get a feeling of Camelot. We assume that the reader has some knowledge of Python.

The reader can read the following sub-sections in any order.

2.1 Creating a Movie Database Application

In this tutorial we will create a fully functional movie database application with Camelot. We assume Camelot is properly *installed*. An all in one installer for Windows is available as an SDK to develop Camelot applications (Python SDK).

2.1.1 Setup Spyder

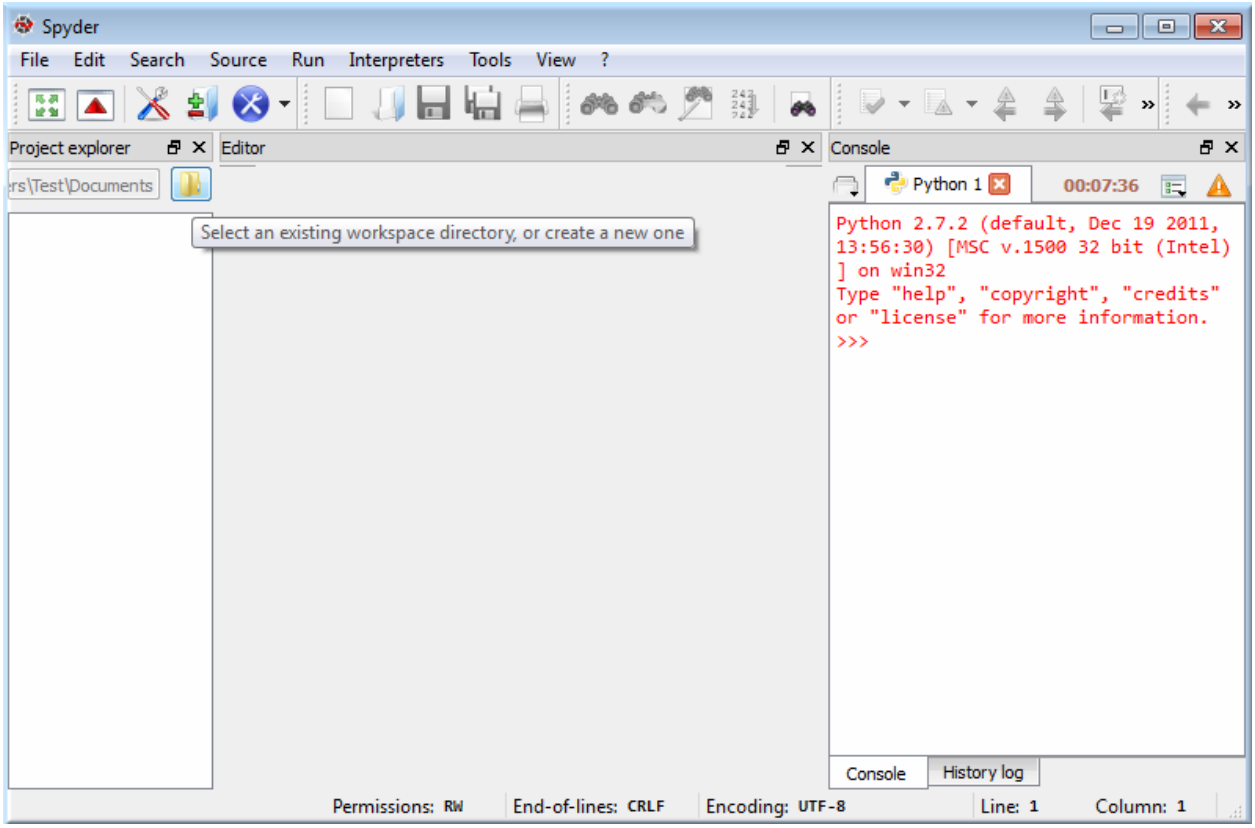
In this section, we will explain how to setup the **Spyder IDE** for developing a **Camelot** project. If you are not using **Spyder**, you can skip this and jump to the next *section*.

Start → *All Programs* → *Python SDK* → *Spyder*

Within **Spyder**, open the *Project Explorer* :

View → *Windows and toolbars* → *Project explorer*

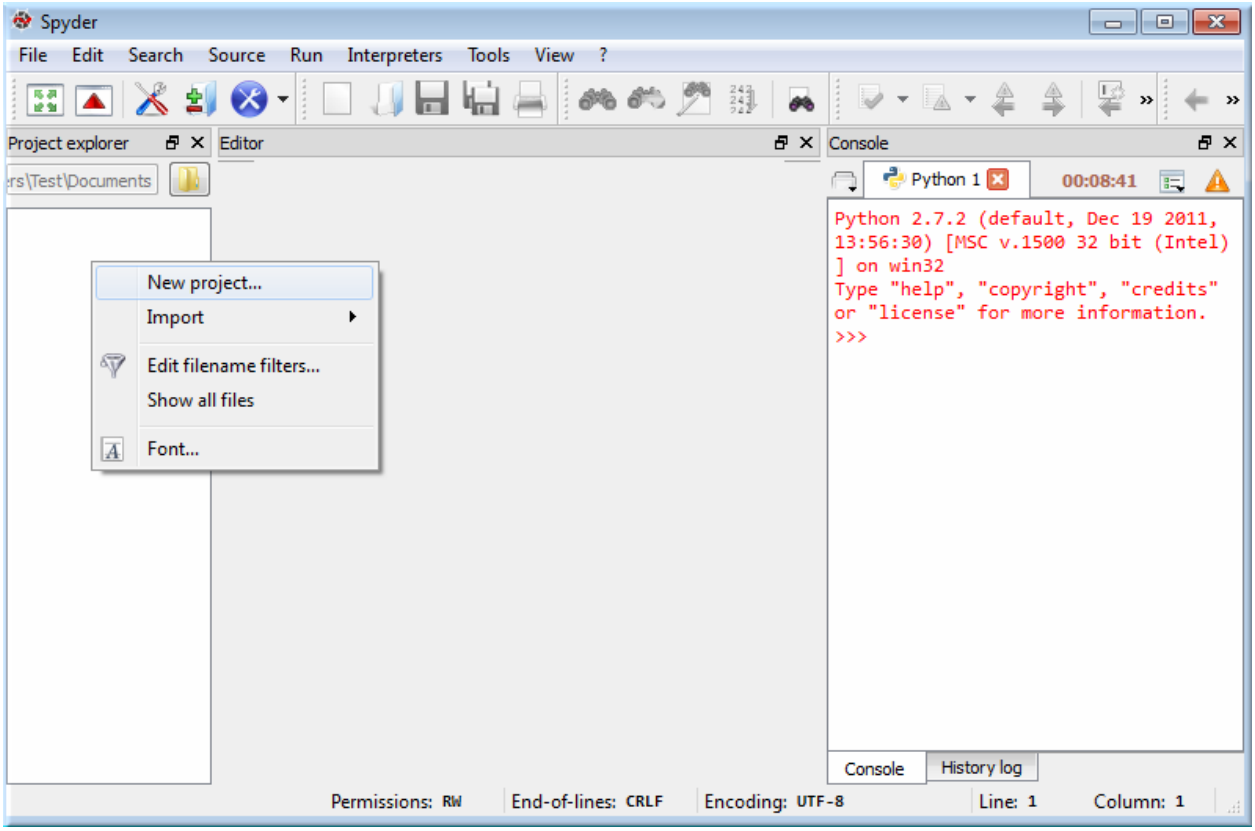
In the *Project Explorer* change the workspace directory, to the directory where you want to put your **Camelot** Projects.



Next, still in the *Project Explorer*, right click to create a new project using :

New Project

Enter *Videostore* as the project name.



2.1.2 Starting a new Camelot project

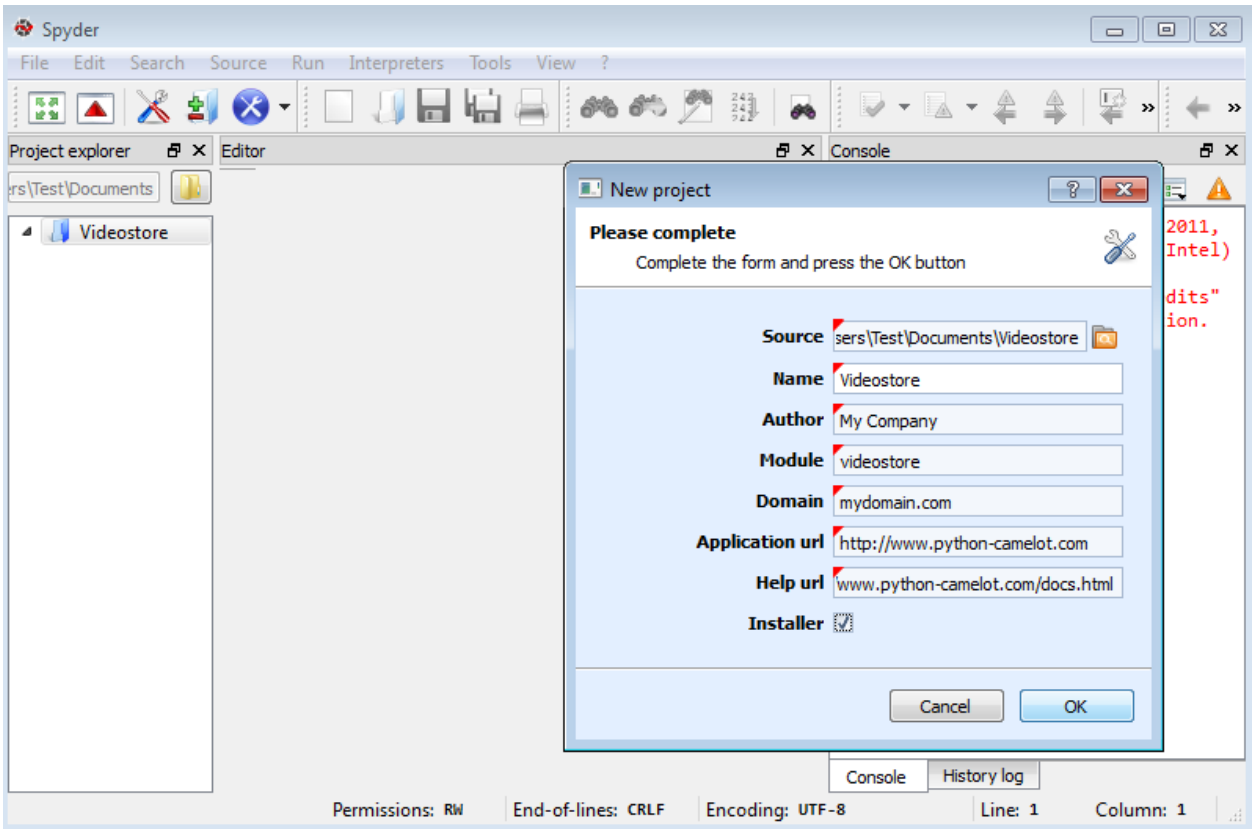
We begin with the creation of a new **Camelot** project, using the *camelot_admin* tool :

Start → All Programs → Python SDK → New Camelot Application

Note: From the command prompt (or shell), go to the directory in which the new project should be created. Type the following command:

```
python -m camelot.bin.camelot_admin
```

A dialog appears where the basic information of the application can be filled in. Select the newly created *Videostore* directory as the location of the source code.



Press *OK* to generate the source code of the project. The source code should now appear in the selected directory.

2.1.3 Main Window and Views

To run the application, double click on the `main.py` file in **Spyder**, which contains the entry point of your **Camelot** application and run this file.

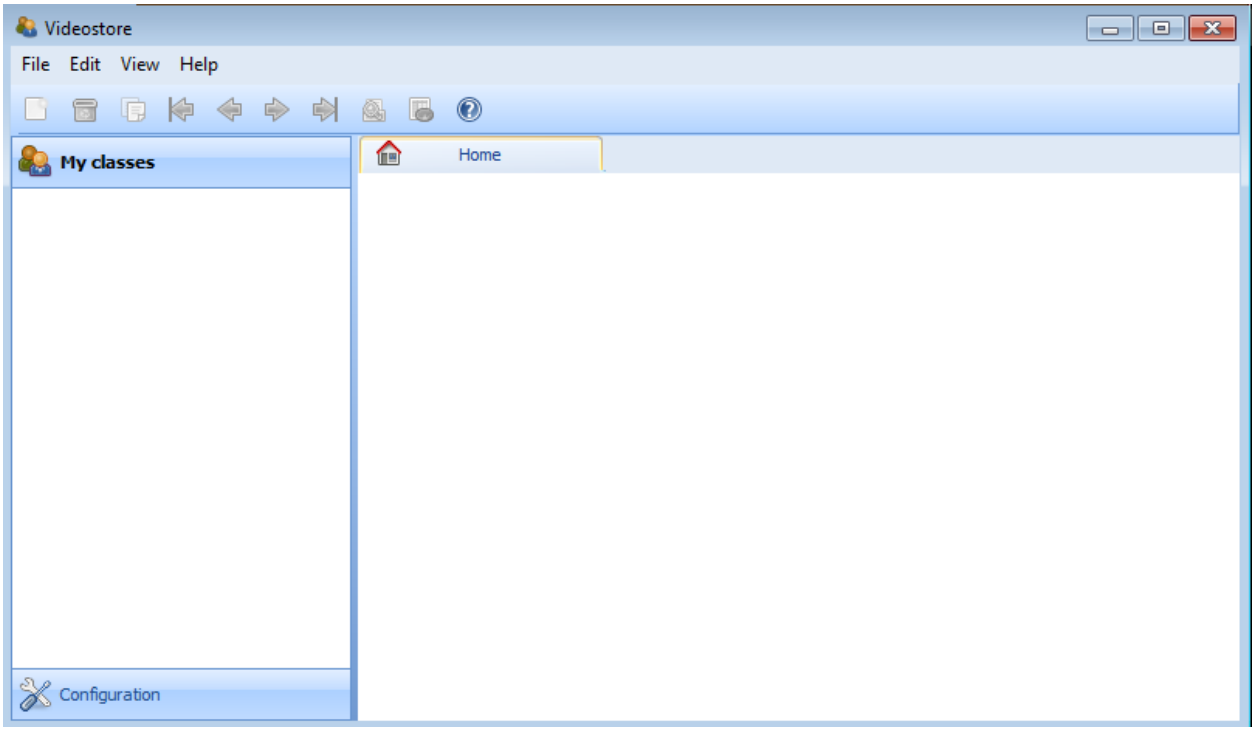
Run → *Run* → *Ok*

Note: From the command prompt, simply start the script

```
python main.py
```

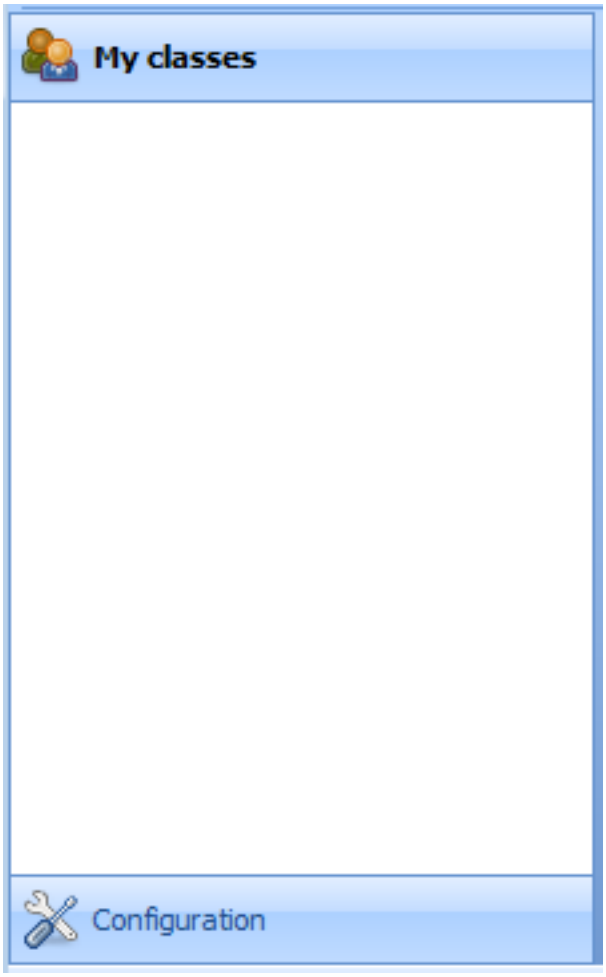
---

your Qt GUI should look like the one we show in the picture below:



The application has a customizable menu and toolbar, a left navigation pane, and a central area, where default the *Home* tab is opened, on which nothing is currently displayed.

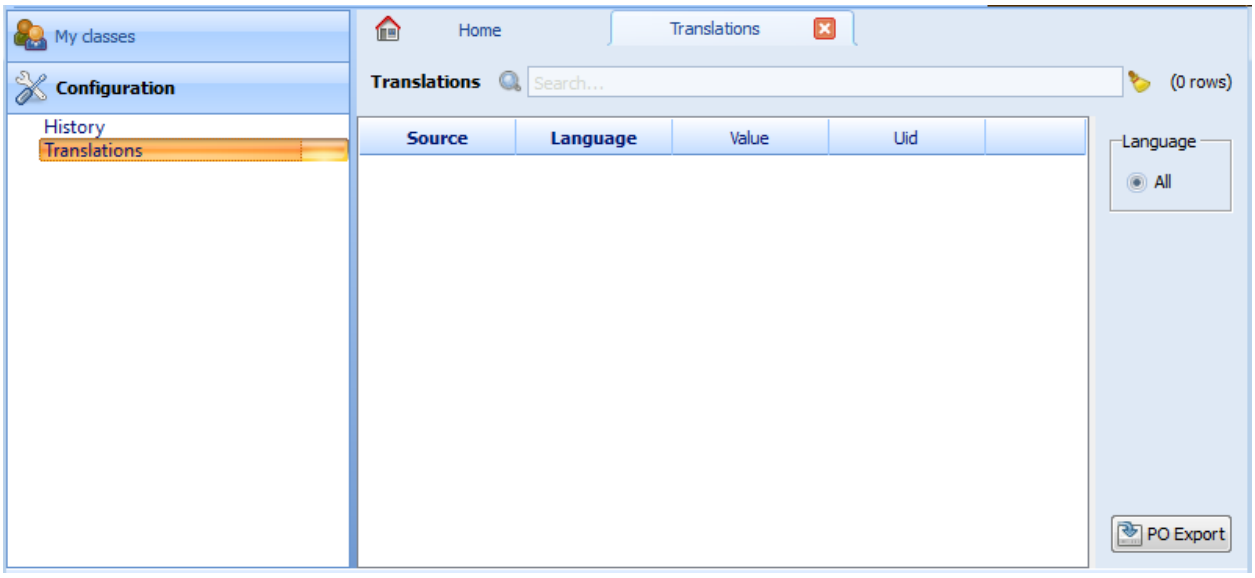
The navigation pane has its first *section* expanded.



The navigation pane uses *Sections* to group *Actions*. Each button in the navigation pane represents a *Section*, and each entry of the navigation tree is an *Action*. Most standard *Actions* open a single table view of an *Entity* in a new tab.

Notice that the application disables most of the menus and the toolbar buttons. When we open a table view, more options become available.

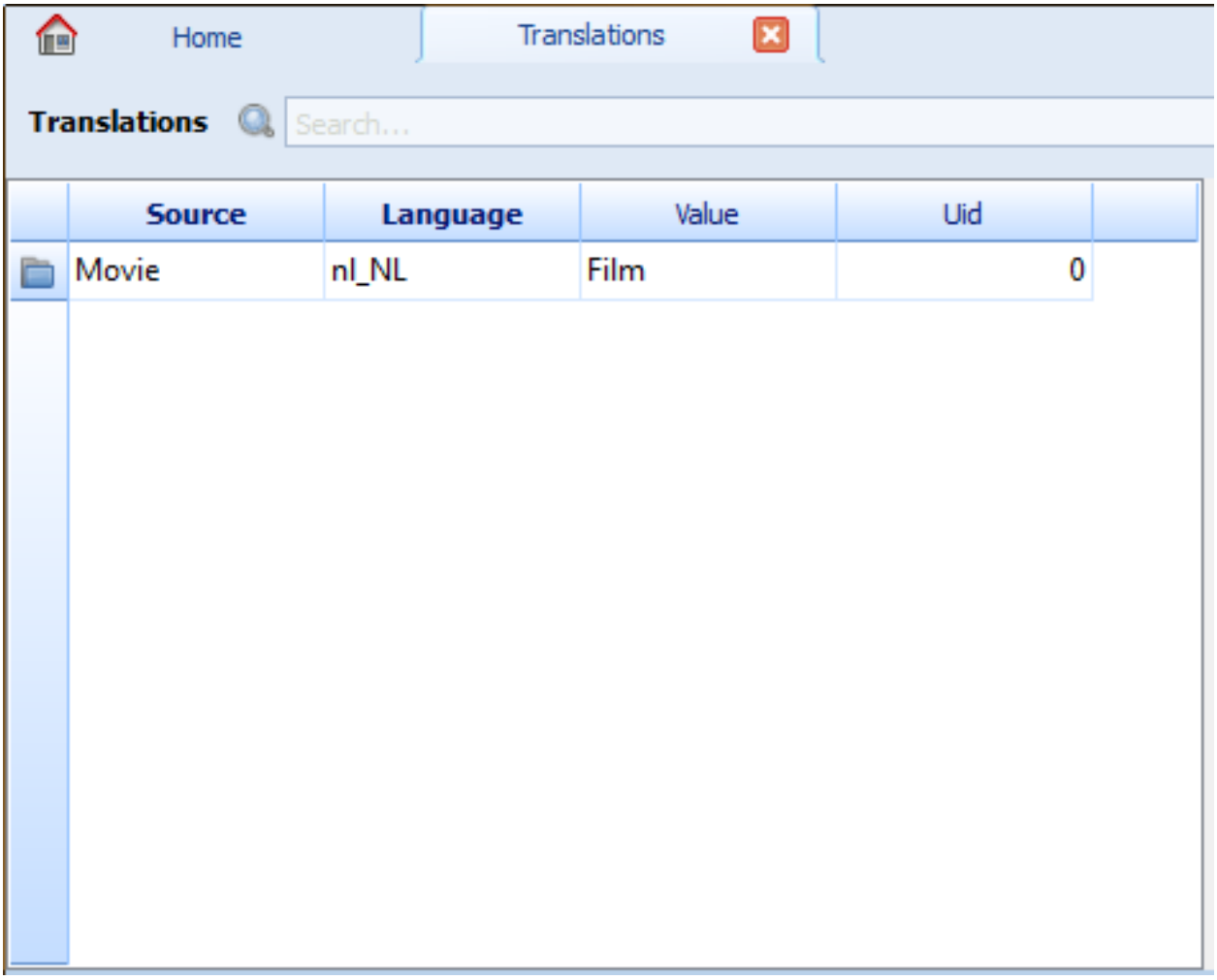
Entities are opened in the active tab, unless they are opened by selecting *Open in New Tab* from the context menu (right click) of the entity link, which will obviously open a new tab to right. Tabs can be closed by clicking the *X* in the tab itself.



Each row is a record with some fields that we can edit (others might not be editable). Let's now add a new row by clicking on the new icon (icon farthest the the left in the toolbar above the navigation pane).

We now see a new window, containing a form view with additional fields. Forms label **required** fields in bold.

Fill in a first and last name, and close the form. Camelot will automatically validate and echo the changes to the database. We can reopen the form by clicking on the blue folder icon in the first column of each row of the table. Notice also that there is now an entry in our table.



That's it for basic usages of the interface. Next we will write code for our database model.

2.1.4 Creating the Movie Model

Let's first take a look at the `main.py` in our project directory. It contains a *my_settings* object which is appended to the global *settings*. The *Global settings* object contains the global configuration for things such as database and file location.

Now we can look at `model.py`. Camelot has already imported some classes for us. They are used to create our entities. Let's say we want a movie entity with a `title`, a short `description`, a `release date`, and a `genre`.

The aforementioned specifications translate into the following Python code, that we add to our `model.py` module:

```
from sqlalchemy import Unicode, Date
from sqlalchemy.schema import Column
from camelot.core.orm import Entity
from camelot.admin.entity_admin import EntityAdmin

class Movie( Entity ):

    __tablename__ = 'movie'

    title = Column( Unicode(60), nullable = False )
```

```
    short_description = Column( Unicode(512) )
    release_date = Column( Date() )
    genre = Column( Unicode(15) )
```

Note: The complete source code of this tutorial can be found in the `camelot_example` folder of the Camelot source code.

`Movie` inherits `camelot.core.orm.Entity`, which is the declarative base class for all objects that should be stored in the database. We use the `__tablename__` attribute to to name the table ourselves in which the data will be stored, otherwise a default tablename would have been used.

Our entity holds four fields that are stored in columns in the table.

```
title = Column( Unicode(60), nullable = False )
```

`title` holds up to 60 unicode characters, and cannot be left empty:

```
short_description = Column( Unicode(512) )
```

`short_description` can hold up to 512 characters:

```
release_date = Column( Date() )
genre = Column( Unicode(15) )
```

`release_date` holds a date, and `genre` up to 15 unicode characters:

For more information about defining models, refer to the SQLAlchemy Declarative extension.

The different SQLAlchemy column types used are described here. Finally, custom Camelot fields are documented in the section *camelot-column-types*.

Let's now create an `EntityAdmin` subclass

2.1.5 The EntityAdmin Subclass

We have to tell Camelot about our entities, so they show up in the GUI (Graphical User Interface). This is one of the purposes of `camelot.admin.entity_admin.EntityAdmin` subclasses. After adding the `EntityAdmin` subclass, our `Movie` class now looks like this:

```
class Movie( Entity ):

    __tablename__ = 'movie'

    title = Column( Unicode(60), nullable = False )
    short_description = Column( Unicode(512) )
    release_date = Column( Date() )
    genre = Column( Unicode(15) )

    def __unicode__( self ):
        return self.title or 'Untitled movie'

    class Admin( EntityAdmin ):
        verbose_name = 'Movie'
        list_display = ['title', 'short_description', 'release_date', 'genre']
```

We made `Admin` an inner class to strengthen the link between it and the `Entity` subclass. Camelot does not force us. Assign your `EntityAdmin` class to the `Admin` `Entity` member to put it somewhere else.

`verbose_name` will be the label used in navigation trees.

The last attribute is interesting; it holds a list containing the fields we have defined above. As the name suggests, `list_display` tells Camelot to only show the fields specified in the list. `list_display` fields are also taken as the default fields to show on a form.

In our case we want to display four fields: `title`, `short_description`, `release_date`, and `genre` (that is, all of them.)

The fields displayed on the form can optionally be specified too in the `form_display` attribute.

We also add a `__unicode__()` method that will return either the title of the movie entity or `'Untitled movie'` if title is empty. The `__unicode__()` method will be called in case Camelot needs a textual representation of an object, such as in a window title.

Let's move onto the last piece of the puzzle.

2.1.6 Configuring the Application

We are now working with `application_admin.py`. One of the tasks of `application_admin.py` is to specify the sections in the left pane of the main window.

The created application has a class, `MyApplicationAdmin`. This class is a subclass of `camelot.admin.application_admin.ApplicationAdmin`, which is used to control the overall look and feel of every Camelot application.

To change sections in the left pane of the main window, simply overwrite the `get_sections` method, to return a list of the desired sections. By default this method contains:

```
def get_sections(self):
    from camelot.model.memento import Memento
    from camelot.model.i18n import Translation
    return [ Section( _('My classes'),
                      self,
                      Icon('tango/22x22/apps/system-users.png'),
                      items = [] ),
             Section( _('Configuration'),
                      self,
                      Icon('tango/22x22/categories/preferences-system.png'),
                      items = [Memento, Translation] )
           ]
```

which will display two buttons in the navigation pane, labelled `'My classes'` and `'Configurations'`, with the specified icon next to each label. And yes, the order matters.

We need to add a new section for our `Movie` entity, this is done by extending the list of sections returned by the `get_sections` method with a Movie section:

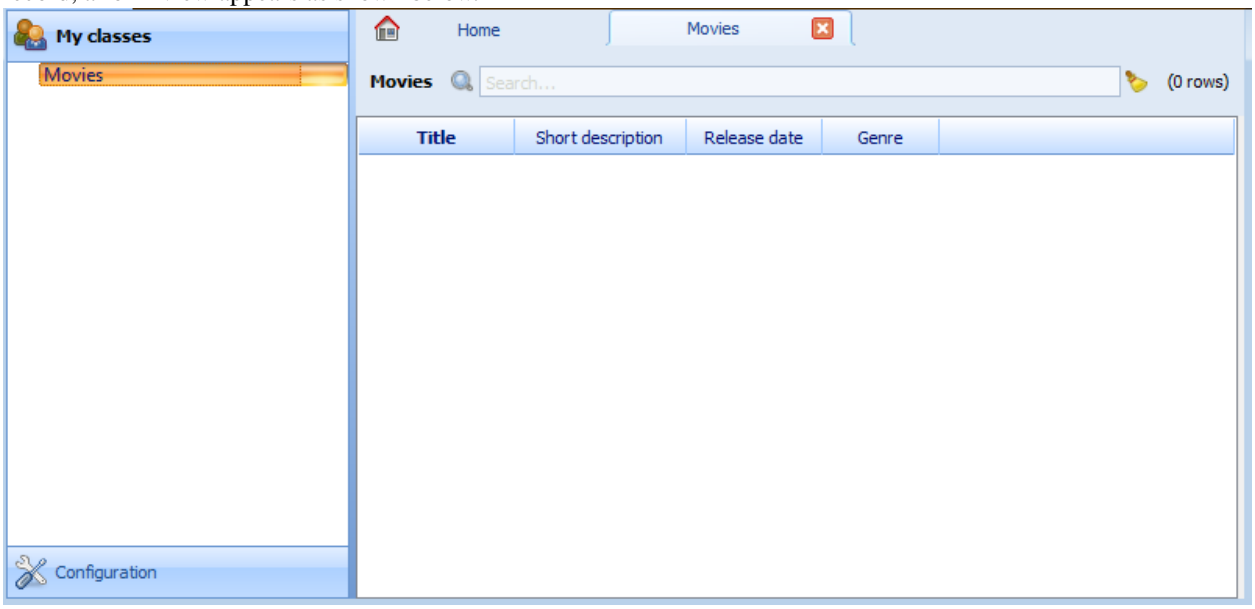
```
from videostore.model import Movie
return [ Section( _('Movie'),
                  self,
                  Icon('tango/22x22/apps/system-users.png'),
                  items = [Movie] ),
         Section( _('Configuration'),
                  self,
                  Icon('tango/22x22/categories/preferences-system.png'),
                  items = [Memento, Translation] )
       ]
```

The constructor of a section object takes the name of the section, a reference to the application admin object, the icon to be used and the items in the section. The items is a list of the entities for which a table view should shown.

Camelot comes with the Tango icon collection; we use a suitable icon for our movie section.

We can now try our application.

We see a new button the navigation pane labelled *'Movies'*. Clicking on it fills the navigation tree with the only entity in the movies's section. Clicking on this tree entry opens the table view. And if we click on the blue folder of each record, a form view appears as shown below.



That's it for the basics of defining an entity and setting it for display in Camelot. Next we look at relationships between entities.

2.1.7 Relationships

We will be using SQLAlchemy's `sqlalchemy.orm.relationship` API. We'll relate a director to each movie. So first we need a `Director` entity. We define it as follows:

```
class Director( Entity ):

    __tablename__ = 'director'

    name = Column( Unicode( 60 ) )
```

Even if we define only the `name` column, Camelot adds an `id` column containing the primary key of the `Director` Entity. It does so because we did not define a primary key ourselves. This primary key is an integer number, unique for each row in the `director` table, and as such unique for each `Director` object.

Next, we add a reference to this primary key in the movie table, this is called the foreign key. This foreign key column, called `director_id` will be an integer number as well, with the added constraint that it can only contain values that are present in the `director` table its `id` column.

Because the `director_id` column is only an integer, we need to add the `director` attribute of type `relationship`. This will allow us to use the `director` property as a `Director` object related to a `Movie` object. The `relationship` attribute will find out about the `director_id` column and use it to attach a `Director` object to a `Movie` object

```
from sqlalchemy.schema import ForeignKey
from sqlalchemy.orm import relationship

class Movie( Entity ):

    __tablename__ = 'movie'

    title = Column( Unicode( 60 ),  nullable = False )
    short_description = Column( Unicode( 512 ) )
    release_date = Column( Date() )
    genre = Column( Unicode( 15 ) )

    director_id = Column( Integer, ForeignKey('director.id') )
    director = relationship( 'Director',
                             backref = 'movies' )

    class Admin( EntityAdmin ):
        verbose_name =  'Movie'
        list_display = [ 'title',
                         'short_description',
                         'release_date',
                         'genre',
                         'director' ]

    def __unicode__( self ):
        return self.title or 'untitled movie'
```

We also inserted `'director'` in `list_display`.

To be able to have the movies accessible from a director, a `backref` is defined in the *director* relationship. This will result in a `movies` attribute for each director, containing a list of movie objects.

Our `Director` entity needs an administration class as well. We will also add `__unicode__()` method as suggested above. The entity now looks as follows:

```
class Director( Entity ):
    __tablename__ = 'director'

    name = Column( Unicode(60) )

    class Admin( EntityAdmin ):
        verbose_name = 'Director'
        list_display = [ 'name' ]
        form_display = list_display + ['movies']

    def __unicode__(self):
        return self.name or 'unknown director'
```

Note: Whenever the model changes, the database needs to be updated. This can be done by hand, or by dropping and recreating the database (or deleting the sqlite file). By default Camelot stores the data in an local directory specified by the operating system. Look in the startup logs to see where they are stored on your system, look for a line like

```
[INFO    ] [camelot.core.conf] - store database and media in /home/username/.camelot/videostore
```

For completeness the two entities are once again listed below:

```
class Movie( Entity ):
```

```
    __tablename__ = 'movie'

    title = Column( Unicode( 60 ), nullable = False )
    short_description = Column( Unicode( 512 ) )
    release_date = Column( Date() )
    genre = Column( Unicode( 15 ) )

    director_id = Column( Integer, ForeignKey('director.id') )
    director = relationship( 'Director',
                             backref = 'movies' )

    class Admin( EntityAdmin ):
        verbose_name =  'Movie'
        list_display = [ 'title',
                         'short_description',
                         'release_date',
                         'genre',
                         'director' ]

    def __unicode__( self ):
        return self.title or 'untitled movie'

class Director( Entity ):
    __tablename__ = 'director'

    name = Column( Unicode(60) )

    class Admin( EntityAdmin ):
        verbose_name = 'Director'
        list_display = [ 'name' ]
        form_display = list_display + ['movies']

    def __unicode__(self):
        return self.name or 'unknown director'
```

The last step is to fix `application_admin.py` by adding the following lines to the Director entity to the Movie section:

```
Section( 'Movies',
         self,
         Icon( 'tango/22x22/mimetypes/x-office-presentation.png' ),
         items = [ Movie, Director ])
```

This takes care of the relationship between our two entities.

We have just learned the basics of Camelot, and have a nice movie database application we can play with. In another tutorial, we will learn more advanced features of Camelot.

2.2 Creating a Report with Camelot

With the Movie Database Application as our starting point, we're going to use the reporting framework in this tutorial. We will create a report of each movie, which we can access from the movie detail page.

2.2.1 Massaging the model

First of all we need to create a button to access our report. This is easily done by specifying a form_action, right in the Admin subclass of the model. Our appended code will be:

```
form_actions = [MovieSummary()]
```

The action is described in the MovieSummary class, which we'll discuss next. Note that it needs to imported, obviously:

```
from movie_summary import MovieSummary
```

So the movie model admin will look like this:

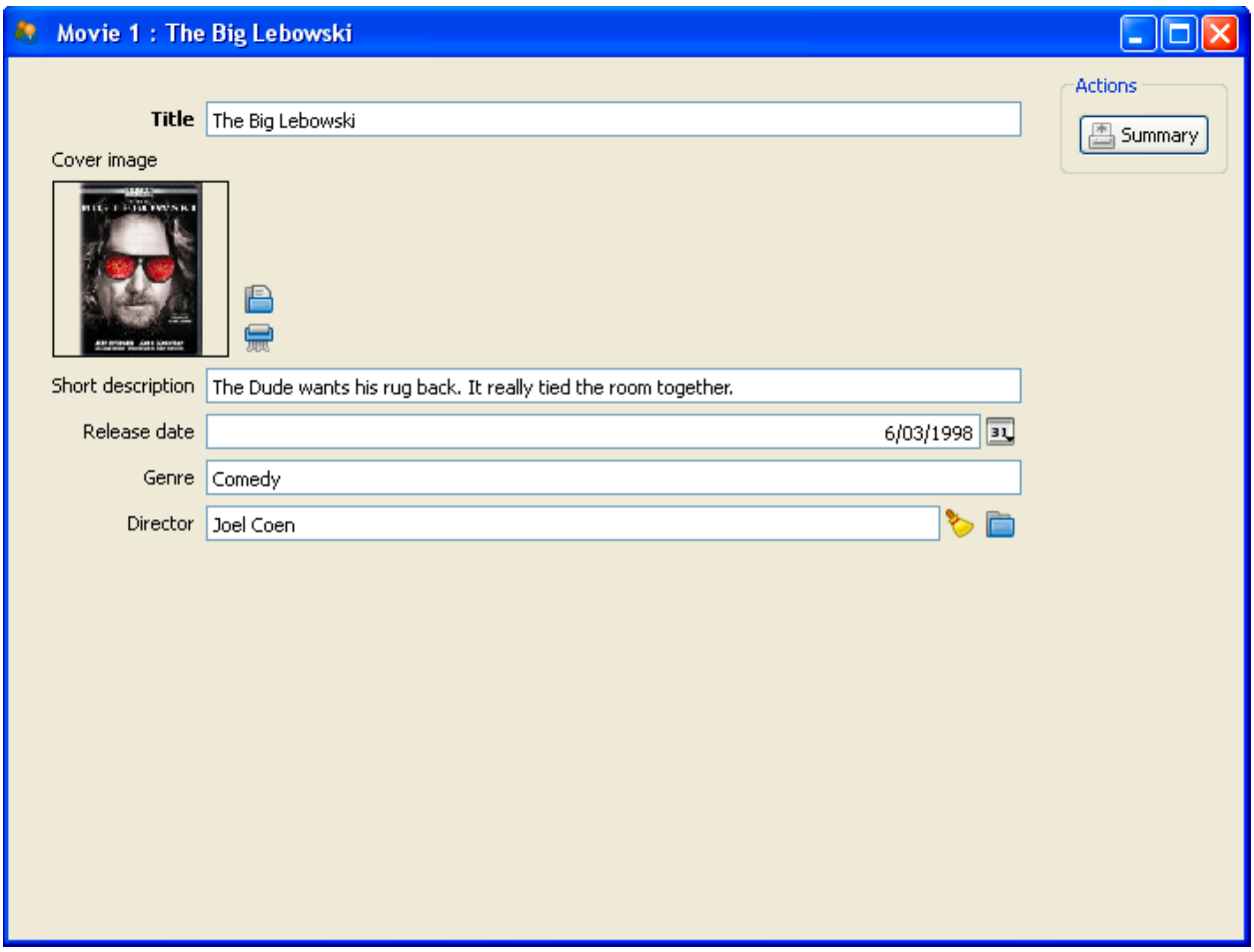
```
class Admin(EntityAdmin):
        from movie_summary import MovieSummary
        verbose_name = _('Movie')
        list_display = [
                'title',
                'short_description',
                'release_date',
                'genre',
                'director'
        ]
        form_display = [
                'title',
                'cover_image',
                'short_description',
                'release_date',
                'genre',
                'director'
        ]
        form_actions = [
                MovieSummary()
        ]
```

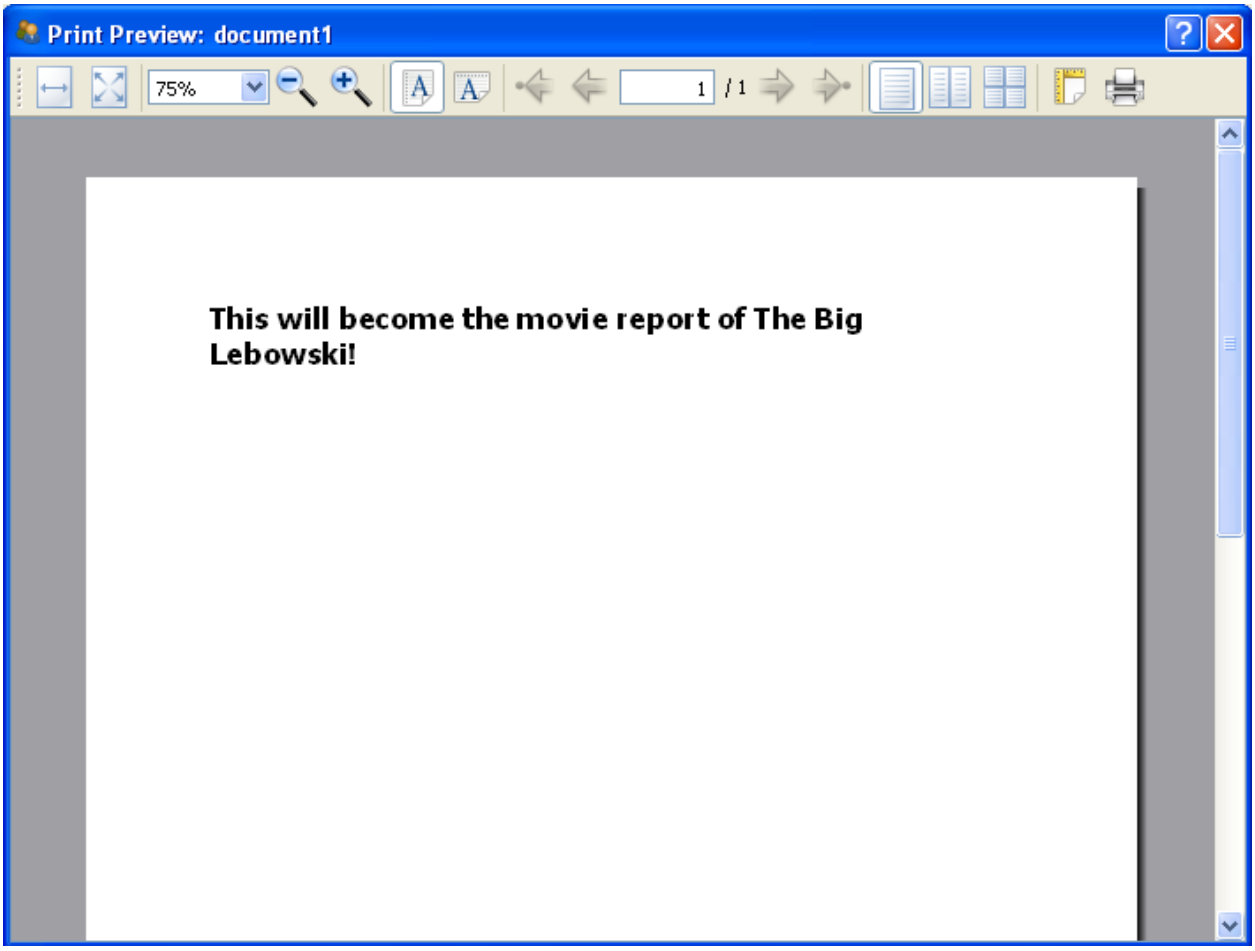
2.2.2 The Summary class

In the MovieSummary class, which is a child class of `camelot.admin.action.base.Action`, we need to override just one method; the `model_run()` method, which has the *model_context* object as its argument. This makes accessing the *Movie* object very easy as we'll see in a minute. The *model_run* method will yield ..., have a guess.... Exactly, a print preview:

```
        class MovieSummary( Action ):

            verbose_name = _('Summary')

            def model_run(self, model_context):
                from camelot.view.action_steps import PrintHtml
                movie = model_context.get_object()
                yield PrintHtml( "<h1>This will become the movie report of %s!</h1>" % movie.title )
```

You can already test this. You should see a button in the "Actions" section, on the right of the Movie detail page. Click this and a print preview should open with the text you let the html method return.


Movie 1 : The Big Lebowski
Actions
Summary
Title
The Big Lebowski
Cover image
Short description
The Dude wants his rug back. It really tied the room together.
Release date
6/03/1998
Genre
Comedy
Director
Joel Coen



Now let's make it a bit fancier.

2.2.3 Using Jinja templates

Install and add Jinja2 to your PYTHONPATH. You can find it here: http://jinja.pocoo.org/2/ or at the cheeseshop http://pypi.python.org/pypi/Jinja2 . Now let's use its awesome powers.

First we'll make a base template. This will determine our look and feel for all the report pages. This is basically html and css with block definitions. Later we'll create the page movie summary template which will contain our model data. The movie summary template will inherit the base template, and provide content for the aforementioned blocks. The base template could look something like:

```
<html>
<head>
  <title>{% block page_head_title %}{% endblock %}</title>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  <style type="text/css">
body, html {
        font-family: Verdana, Arial, sans-serif;
}
{% block styles %}{% endblock %}
  </style>
</head>
<body>
```

```
<table id="page_header" width="100%">
  <tr>
    <td><h1>{% block page_header %}{% endblock %}</h1></td>
    <td align="right">{% block page_header_right %}{% endblock %}</td>
  </tr>

</table>
<hr>
<h2 id="page_title"><center>{% block page_title %}{% endblock %}</center></h2>
<hr>
{% block page_content %}{% endblock %}
<hr>
<div id="page_footer">{% block page_footer %}{% endblock %}</div>

</body>
</html>
```

We'll save this file as base.html in a directory called templates in our videostore. Like this base template, the movie summary template is html and css. Take a look at the example first:

```
{% extends 'base.html' %}
{% block styles %}{{ style }}{% endblock %}
{% block page_head_title %}{{ title }}{% endblock %}
{% block page_title %}{{ title }}{% endblock %}
{% block page_header %}{{ header }}{% endblock %}
{% block page_header_right %}
{% if cover %}
        <img src="{{ cover }}" alt="">
{% else %}
        (no cover)
{% endif %}
{% endblock %}
{% block page_content %}{{ content }}{% endblock %}
{% block page_footer %}{{ footer }}{% endblock %}
```

First we extend the base template, that way we don't need to worry about the boilerplate stuff, and keep our pages consistent, provided we create more reports of course. We can now fill in the blanks, erm blocks from the base template. We do that with placeholders which we'll define in the html method of our MovieSummary class. This way we can even add style to the page:

```
{% block styles %}{{ style }}{% endblock %}
```

We'll define this later. The templating language also allows basic flow control:

```
{% if cover %}
        <img src="{{ cover }}" alt="">
{% else %}
        (no cover)
{% endif %}
```

If there is no cover image, we'll show the string "(no cover)". We'll save this file as movie_summary.html in the templates directory.

Like i said earlier, we now need to define which values will go in the placeholders, so let's update our html method in the MovieSummary class. First, we import the needed elements:

```
import datetime
from jinja import Environment, FileSystemLoader
from pkg_resources import resource_filename
```

```
import videostore
from camelot.core.conf import settings
```

We'll be printing a date, so we'll need datetime. The Jinja classes to make use of our templates. And to locate our templates, we'll use the resource module, with our videostore. And load up the Jinja environment ...

```
fileloader = FileSystemLoader(resource_filename(videostore.__name__, 'templates'))
e = Environment(loader=fileloader)
```

Now we need to create a context dictionary to provide data to the templates. The keys of this dictionary are the placeholders we used in our movie_summary template, the values we can use from the model, which is passed as the o argument in the html method:

```
context = {
'header':o.title,
'title':'Movie Summary',
'style':'.label { font-weight:bold; }',
'content':'<span class="label">Description:</span> %s<br>\
        <span class="label">Release date:</span> %s<br>\
        <span class="label">Genre:</span> %s<br>\
        <span class="label">Director:</span> %s'
        % (o.short_description, o.release_date, o.genre, o.director),
'cover': os.path.join( settings.CAMELOT_MEDIA_ROOT(), 'covers', o.cover_image.name ),
'footer':'<br>copyright %s - Camelot' % datetime.datetime.now().year
}
```

Plain old Python dictionary. Check it out, we can even pass css in our setup.

Finally, we'll get the template from the Jinja environment and return the rendered result of our context:

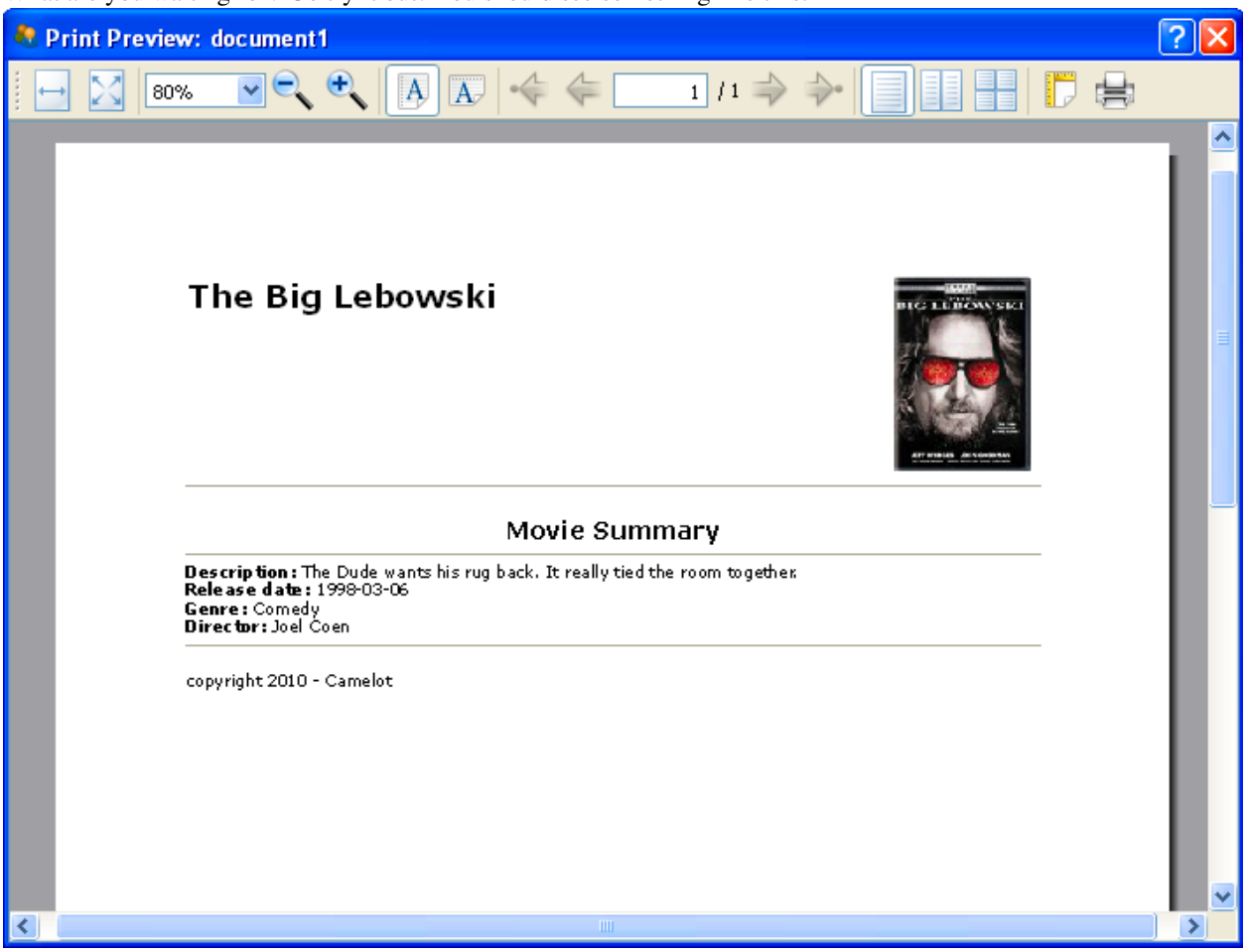
```
t = e.get_template('movie_summary.html')
return t.render(context)
```

So our finished method eventually looks like this:

```
from camelot.admin.action import Action

class MovieSummary( Action ):

    verbose_name = _('Summary')

    def model_run( self, model_context ):
        from camelot.view.action_steps import PrintHtml
        import datetime
        import os
        from jinja import Environment, FileSystemLoader
        from pkg_resources import resource_filename
        import videostore
        from camelot.core.conf import settings

        fileloader = FileSystemLoader(resource_filename(videostore.__name__, 'templates'))
        e = Environment(loader=fileloader)
        movie = model_context.get_object()
        context = {
                'header':movie.title,
                'title':'Movie Summary',
                'style':'.label { font-weight:bold; }',
                'content':'<span class="label">Description:</span> %s<br>\
                        <span class="label">Release date:</span> %s<br>\
                        <span class="label">Genre:</span> %s<br>\
```

```
                    <span class="label">Director:</span> %s'
                    % (movie.short_description, movie.release_date, movie.genre, movie.director),
            'cover': os.path.join( settings.CAMELOT_MEDIA_ROOT(), 'covers', movie.cover_image.nam
            'footer':'<br>copyright %s - Camelot' % datetime.datetime.now().year
    }
    t = e.get_template('movie_summary.html')
    yield PrintHtml( t.render(context) )
```

What are you waiting for? Go try it out! You should see something like this:



2.3 Add an import wizard to an application

In this tutorial we will add an import wizard to the movie database application created in the *Creating a Movie Database Application* tutorial.

We assume Camelot is properly *installed* and the movie database application is working.

_static/controls/main_window.png

2.3.1 Introduction

Most applications need a way to import data. This data is often delivered in files generated by another application or company. To demonstrate this process we will build a wizard that allows the user to import cover images into the movie database. For each image the user selects, a new Movie will be created with the selected image as a cover image.

2.3.2 Create an action

All user interaction in Camelot is handled through *Actions*. For actions that run in the context of the application, we use the *Application Actions*. We first create a file `importer.py` in the same directory as `application_admin.py`.

In this file we create subclass of `camelot.admin.action.Action` which will be the entry point of the import wizard:

```
from camelot.admin.action import Action
from camelot.core.utils import ugettext_lazy as _

class ImportCovers( Action ):
    verbose_name = _('Import cover images')

    def model_run( self, model_context ):
        yield
```

So now we haven an `ImportCovers` action. Such an action has a `verbose_name` class attribute with the name of the action as shown to the user.

The most important method of the action is the `model_run` method, which will be triggered when the user clicks the action. This method should be a generator that yields an object whenever user interaction is required. Everything that happens inside the `model_run` method happens in a different thread than the GUI thread, so it will not block the GUI.

2.3.3 Add the action to the GUI

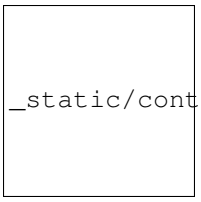
Now the user needs to be able to trigger the action. We edit the `application_admin.py` file and make sure the `ImportCoversAction` is imported.

```
        from camelot_example.importer import ImportCovers
```

Then we add an instance of the `ImportCovers` action to the sections defined in the `get_sections` method of the `ApplicationAdmin`:

```
                Section( _('Movies'),
                         self,
                         Icon('tango/22x22/mimetypes/x-office-presentation.png'),
                         items = [ Movie,
                                   Tag,
                                   VisitorReport,
#                                   VisitorsPerDirector,
                                   ImportCovers() ]),
```

This will make sure the action pops up in the **Movies** section of the application.



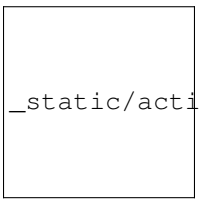
2.3.4 Select the files

To make the action do something useful, we will implement its `model_run` method. Inside the `model_run` method, we can `yield` various `camelot.admin.action.base.ActionStep` objects to the GUI. An `ActionStep` is a part of the action that requires user interaction (the user answering a question). The result of this interaction is returned by the `yield` statement.

To ask the user for a number of image files to import, we will pop up a file selection dialog inside the `model_run` method:

```
    def model_run( self, model_context ):
        from camelot.view.action_steps import ( SelectFile,
                                                UpdateProgress,
                                                Refresh,
                                                FlushSession )

        select_image_files = SelectFile( 'Image Files (*.png *.jpg);;All Files (*)' )
        select_image_files.single = False
        file_names = yield select_image_files
        file_count = len( file_names )
```

The `yield` statement returns a list of file names selected by the user.



2.3.5 Create new movies

First make sure the `Movie` class has an `camelot.types.Image` field named `cover` which will store the image files.

```
    cover = Column( camelot.types.Image( upload_to = 'covers' ) )
```

Next we add to the `model_run` method the actual creation of new movies.

```
        import os
        from sqlalchemy import orm
        from camelot.core.orm import Session
        from camelot_example.model import Movie

        movie_mapper = orm.class_mapper( Movie )
        cover_property = movie_mapper.get_property( 'cover' )
        storage = cover_property.columns[0].type.storage
        session = Session()
```

```
        for i, file_name in enumerate(file_names):
            yield UpdateProgress( i, file_count )
            title = os.path.splitext( os.path.basename( file_name ) )[0]
            stored_file = storage.checkin( unicode( file_name ) )
            movie = Movie( title = unicode( title ) )
            movie.cover = stored_file

        yield FlushSession( session )
```

In this part of the code several things happen :

Store the images

In the first lines, we do some sqlalchemy magic to get access to the `storage` attribute of the `cover` field. This `storage` attribute is of type `camelot.core.files.storage.Storage`. The `Storage` represents the files managed by Camelot.

Create Movie objects

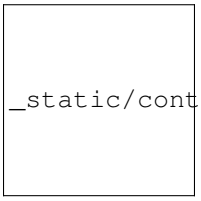
Then for each file, a new `Movie` object is created with as title the name of the file. For the `cover` attribute, the file is checked in into the `Storage`. This actually means the file is copied from its original directory to a directory managed by Camelot.

Write to the database

In the last line, the `session` is flushed and thus all changes are written to the database. The `camelot.view.action_steps.orm.FlushSession` action step flushes the session and propagetes the changes to the GUI.

Keep the user informed

For each movie imported, a `camelot.view.action_steps.update_progress.UpdateProgress` object is `yield` to the GUI to inform the user of the import progress. Each time such an object is yielded, the progress bar is updated.



2.3.6 Refresh the GUI

The last step of the `model_run` method will be to refresh the GUI. So if the user has the `Movies` table open when importing, this table will show the newly created movies.

```
        yield Refresh()
```

2.3.7 Result

This is how the resulting `importer.py` file looks like :

```
from camelot.admin.action import Action
from camelot.core.utils import ugettext_lazy as _
from camelot.view.art import Icon
```

```
class ImportCovers( Action ):
    verbose_name = _('Import cover images')
    icon = Icon('tango/22x22/mimetypes/image-x-generic.png')

# begin select files
    def model_run( self, model_context ):
        from camelot.view.action_steps import ( SelectFile,
                                                UpdateProgress,
                                                Refresh,
                                                FlushSession )

        select_image_files = SelectFile( 'Image Files (*.png *.jpg);;All Files (*)' )
        select_image_files.single = False
        file_names = yield select_image_files
        file_count = len( file_names )
# end select files
# begin create movies
        import os
        from sqlalchemy import orm
        from camelot.core.orm import Session
        from camelot_example.model import Movie

        movie_mapper = orm.class_mapper( Movie )
        cover_property = movie_mapper.get_property( 'cover' )
        storage = cover_property.columns[0].type.storage
        session = Session()

        for i, file_name in enumerate(file_names):
            yield UpdateProgress( i, file_count )
            title = os.path.splitext( os.path.basename( file_name ) )[0]
            stored_file = storage.checkin( unicode( file_name ) )
            movie = Movie( title = unicode( title ) )
            movie.cover = stored_file

        yield FlushSession( session )
# end create movies
# begin refresh
        yield Refresh()
# end refresh
```

2.3.8 Unit tests

Once an action works, its important to keep it working as the development of the application continues. One of the advantages of working with generators for the user interaction, is that its easy to simulate the user interaction towards the `model_run()` method of the action. This is done by using the `send()` method of the generator that is returned when calling `model_run()` :

```
    def test_example_application_action( self ):
        from camelot_example.importer import ImportCovers
        from camelot_example.model import Movie
        # count the number of movies before the import
        movies = Movie.query.count()
        # create an import action
        action = ImportCovers()
        generator = action.model_run( None )
        select_file = generator.next()
        self.assertFalse( select_file.single )
```

```
        # pretend the user selected a file
        generator.send( [os.path.join( os.path.dirname(__file__), '..', 'camelot_example', 'media', '
        # continue the action till the end
        list( generator )
        # a movie should be inserted
        self.assertEqual( movies + 1, Movie.query.count() )
```

2.3.9 Conclusion

We went through the basics of the action framework Camelot :

- Subclassing a `camelot.admin.action.Action` class
- Implementing the `model_run` method
- `yield` `camelot.admin.action.base.ActionStep` objects to interact with the user
- Add the `camelot.admin.action.base.Action` object to a `camelot.admin.section.Section` in the side pane

More `camelot.admin.action.base.ActionStep` classes can be found in the `camelot.view.action_steps` module.

CHAPTER

THREE

CAMELOT DOCUMENTATION

This is the reference documentation for developing projects using the Camelot library. The first time Camelot developer is encouraged to read *Creating models* and *Admin classes*.

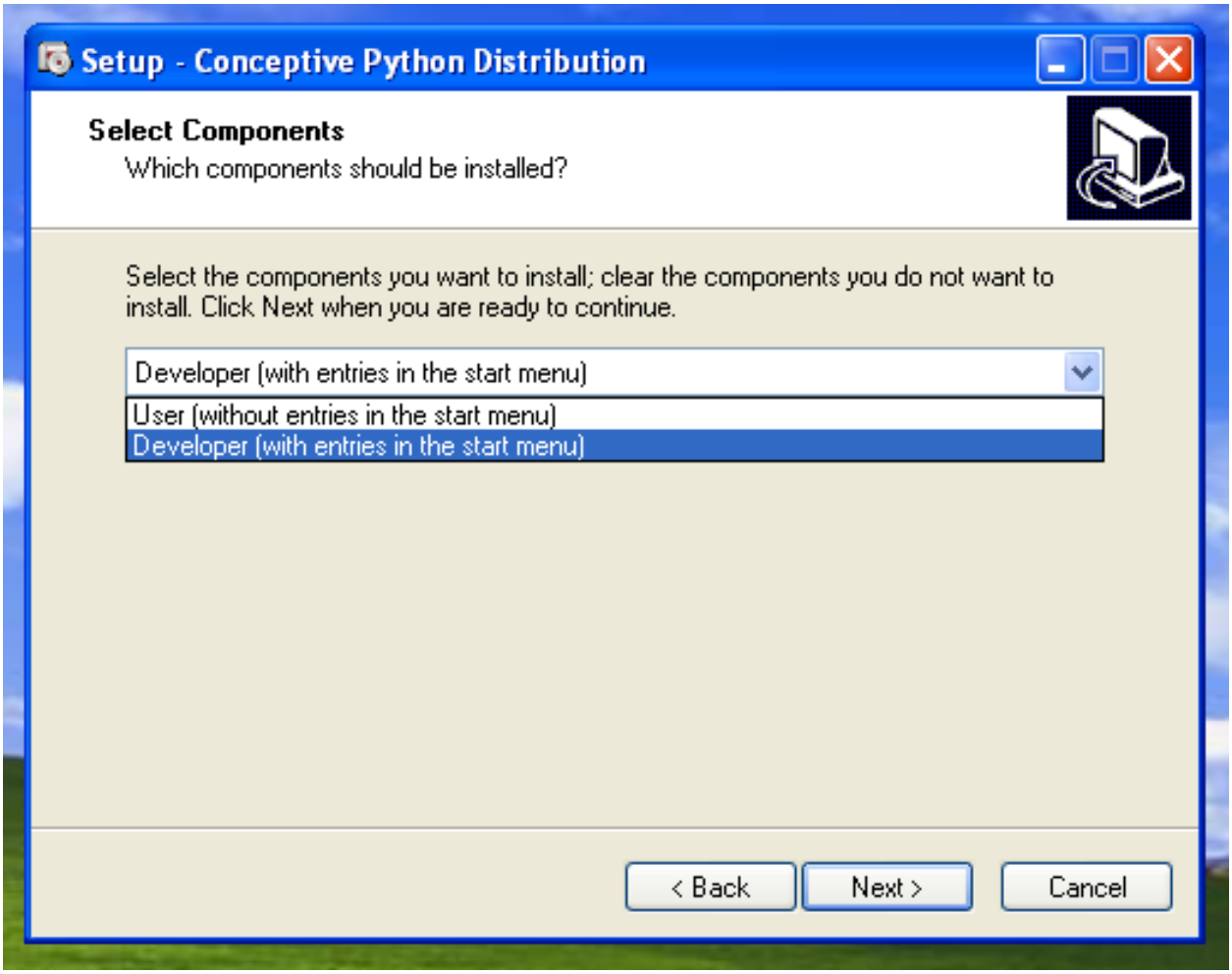
The section *The Two Threads* is for developers whishing to maintain a responsive UI when faced with significant delays in their application code.

All other sections can be read on an as needed base.

3.1 Camelot Installation

3.1.1 All in one Windows installer

When working on Windows, the easiest way to get up and running is through the Conceptive Python SDK.



This SDK is a Python distribution targeted at the development and deployment of QT based applications. This all in one installation of Camelot with all its dependencies is available in the shop.

3.1.2 From the Python Package Index

First, make sure you have setup tools installed, Setup tools. If you are using a debian based distribution, you can type:

```
sudo apt-get install python-setuptools
```

Then use easy_install to install Camelot, under Linux this would be done by typing:

```
sudo easy_install camelot
```

3.1.3 Packages

Linux distributions often offer packages for various applications, including Camelot and its dependencies :

- OpenSUSE build service.

3.1.4 From source

When installing Camelot from source, you need to make sure all dependencies are installed and available in your **PYTHONPATH**.

Dependencies

In addition to PyQt 4.8 and Qt 4.8, Camelot needs these libraries :

SQLAlchemy==0.8.0 Jinja2==2.6 chardet==2.1.1 xlwt==0.7.4 xlrd==0.9.0

Releases

The source code of a release can be downloaded from the Python Package Index and then extracted:

```
tar xzvf Camelot-10.07.02.tar.gz
```

Repository

The latest and greatest version of the source can be checked out from the Bitbucket repository:

```
hg clone https://bitbucket.org/conceptive/camelot
```

Adapting PYTHONPATH

You need to make sure Camelot and all its dependencies are in the **PYTHONPATH** before you start using it.

3.1.5 Verifiy the installation

To verify if you have Camelot installed and available in the **PYTHONPATH**, fire up a python interpreter:

```
python
```

and issue these commands:

```
>>> import camelot
>>> print camelot.__version__
>>> import sqlalchemy
>>> print sqlalchemy.__version__
>>> import PyQt4
```

None of them should raise an ImportError.

3.2 Creating models

Camelot makes it easy to create views for any type of *Python* objects.

SQLAlchemy is a very powerful Object Relational Mapper (ORM) with lots of possibilities for handling simple or sophisticated datastructures. The SQLAlchemy website has extensive documentation on all these features. An important part of Camelot is providing an easy way to create views for objects mapped through SQLAlchemy.

SQLAlchemy comes with the Declarative extension to make it easy to define an ORM mapping using the Active Record Pattern. This is used through the documentation and in the example code.

To use *Declarative*, threre are some base classes that should be imported:

```
from camelot.core.orm import Entity
from camelot.admin.entity_admin import EntityAdmin

from sqlalchemy import sql
```

```
from sqlalchemy.schema import Column
import sqlalchemy.types
```

Those are :

- `camelot.core.orm.Entity` is the declarative base class provided by Camelot for all classes that are mapped to the database, and is a subclass of `camelot.core.orm.entity.EntityBase`
- `camelot.admin.entity_admin.EntityAdmin` is the base class that describes how an *Entity* subclass should be represented in the GUI
- `sqlalchemy.schema.Column` describes a column in the database and a field in the model
- `sqlalchemy.types` contains the various column types that can be used

Next a model can be defined:

```
class Tag(Entity):

    __tablename__ = 'tags'

    name = Column( sqlalchemy.types.Unicode(60), nullable = False )
    movies = ManyToMany( 'Movie',
                         tablename = 'tags_movies__movies_tags',
                         local_colname = 'movies_id',
                         remote_colname = 'tags_id' )

    def __unicode__( self ):
        return self.name

    class Admin( EntityAdmin ):
        form_size = (400,200)
        list_display = ['name']

# begin visitor report definition
```

The code above defines the model for a *Tag* class, an object with only a name that can be related to other ojbects later on. This code has some things to notice :

- *Tag* is a subclass of `camelot.core.orm.Entity`,
- the `__tablename__` class attribute allows us to specify the name of the table in the database in which the tags will be stored.
- The `sqlalchemy.schema.Column` statement add fields of a certain type, in this case `sqlalchemy.types.Unicode`, to the *Tag* class as well as to the *tags* table
- The `__unicode__` method is implemented, this method will be called within Camelot whenever a textual representation of the object is needed, eg in a window title or a many to one widget. It's good practice to always implement the `__unicode__` method for all *Entity* subclasses.

When a new Camelot project is created, the *camelot-admin* tool creates an empty `models.py` file that can be used as a place to start the model definition.

3.2.1 Column types

SQLAlchemy comes with a set of column types that can be used. These column types will trigger the use of a certain `QtGui.QDelegate` to visualize them in the views. Camelot extends those SQLAlchemy field types with some of its own.

An overview of field types from SQLAlchemy and Camelot is given in the table below :

All SQLAlchemy field types can be found in the `sqlalchemy.types` module. All additional Camelot field types can be found in the `camelot.types` module.

3.2.2 Relations

SQLAlchemy uses the *relationship* function to define relations between classes. This function can be used within Camelot as well.

On top of this, Camelot provides some construct in the `camelot.core.orm.relationships` that make setting up relationships a bit easier.

3.2.3 Calculated Fields

To display fields in forms that are not stored into the database but, are calculated at run time, two main options exist. Either those fields are calculated within the database or they are calculated by Python. Normal Python properties can be used to do the calculation in Python, whereas ColumnProperties can be used to do the logic in the database.

Python properties as fields

Normal python properties can be used as fields on forms as well. In that case, there will be no introspection to find out how to display the property. Therefore the delegate (*Specifying delegates*) attribute should be specified explicitly.

```
import math

from camelot.admin.object_admin import ObjectAdmin
from camelot.view.controls import delegates

class Coordinate( object ):

  def __init__( self, x = 0, y = 0 ):
    self.id = 1
    self.x = x
    self.y = y

  @property
  def r( self ):
    return math.sqr( self.x**2, self.y**2 )

  class Admin( ObjectAdmin ):
    form_display = ['x', 'y', 'r']
    field_attributes = dict( x = dict( delegate = delegates.FloatDelegate,
                                       editable = True ),
                             y = dict( delegate = delegates.FloatDelegate,
                                       editable = True ),
                             r = dict( delegate = delegates.FloatDelegate ) )
```

By default, python properties are read-only. They have to be set to editable through the field attributes to make them writeable by the user.

Properties are also used to summarize information from multiple attributes and put them in a single field.

Cascading field changes

Whenever the value of a field is changed, this change can cascade through the model by using properties to manipulate the field instead of manipulating it directly. The example below demonstrates how the value of y should be chopped when x is changed.

```
from camelot.admin.object_admin import ObjectAdmin
from camelot.view.controls import delegates

class Coordinate(object):

  def __init__(self):
    self.id = 1
    self.x = 0.0
    self.y = 0.0

  def _get_x(self):
    return self.x

  def _set_x(self, x):
    self.x = x
    self.y = max(self.y,x)

  _x = property(_get_x, _set_x)

  class Admin(ObjectAdmin):
    form_display = ['_x', 'y',]
    field_attributes = dict(_x=dict(delegate=delegates.FloatDelegate, name='x'),
                            y=dict(delegate=delegates.FloatDelegate),)
    form_size = (100,100)
```

Fields calculated by the database

Having certain summary fields of your models filled by the database has the advantage that the heavy processing is moved from the client to the server. Moreover if the summary builds on information in related records, having the database build the summary reduces the need to transfer additional data from the database to the server.

To display fields in the table and the form view that are the result of a calculation done by the database, a `camelot.core.orm.properties.ColumnProperty` needs to be defined in the Declarative model. In this column property, the sql query can be defined using SQLAlchemy statements. In this example, the *Movie* class gains the *total_visitors* attribute which contains the sum of all visitors that went to a movie.

```
@ColumnProperty
def total_visitors( self ):
    return sql.select( [sql.func.sum( VisitorReport.visitors) ],
                                      VisitorReport.movie_id == self.id )
```

It's important to notice that the value of this field is calculated when the object is fetched from the database. When the user presses F9, all data in the application is refreshed from the database, and thus all column properties are recalculated.

3.2.4 Views

Traditionally, in database land, **views** are queries defined at the database level that act like read-only tables. They allow reuse of common queries across an application, and are very suitable for reporting.

Using **SQLAlchemy** this traditional approach can be used, but a more dynamic approach is possible as well. We can map arbitrary queries to an object, and then visualize these objects with **Camelot**.

The model to start from

In the example movie project, we can take three parts of the model : Person, Movie and VisitorReport:

```
class Person( Party ):
    """Person represents natural persons
    """
    using_options( tablename = 'person' )
    party_id = Field( camelot.types.PrimaryKey(),
                      ForeignKey('party.id'),
                      primary_key = True )
    __mapper_args__ = {'polymorphic_identity': u'person'}
    first_name = Field( Unicode( 40 ), required = True )
    last_name = Field( Unicode( 40 ), required = True )
```

There is a relation between Person and Movie through the director attribute:

```
class Movie( Entity ):

    __tablename__ = 'movies'

    title = Column( sqlalchemy.types.Unicode(60), nullable = False )
    short_description = Column( sqlalchemy.types.Unicode(512) )
    releasedate = Column( sqlalchemy.types.Date )
    genre = Column( sqlalchemy.types.Unicode(15) )
    rating = Column( camelot.types.Rating() )
    #
    # All relation types are covered with their own editor
    #
    director = ManyToOne('Person')
    cast = OneToMany('Cast')
    visitor_reports = OneToMany('VisitorReport', cascade='delete')
    tags = ManyToMany('Tag',
                      tablename = 'tags_movies__movies_tags',
                      local_colname = 'tags_id',
                      remote_colname = 'movies_id' )
```

And a relation between Movie and VisitorReport:

```
class VisitorReport(Entity):

    __tablename__ = 'visitor_report'
```

```
    date = Column( sqlalchemy.types.Date,
                   nullable = False,
                   default = datetime.date.today )
    visitors = Column( sqlalchemy.types.Integer,
                       nullable = False,
                       default = 0 )
    movie = ManyToOne( 'Movie', required = True )
```

Definition of the view

Suppose, we now want to display a table with the total numbers of visitors for all movies of a director.

We first define a plain old Python class that represents the expected results :

```
class VisitorsPerDirector(object):

    class Admin(EntityAdmin):
        verbose_name = _('Visitors per director')
        list_display = table.Table( [ table.ColumnGroup( _('Name and Visitors'), ['first_name', 'last
                                      table.ColumnGroup( _('Official'), ['birthdate', 'social_securit
                                    )
# end column group
```

Then define a function that maps the query that calculates those results to the plain old Python object :

```
def setup_views():
    from sqlalchemy.sql import select, func, and_
    from sqlalchemy.orm import mapper

    from camelot.model.party import Person
    from camelot_example.model import Movie, VisitorReport

    s = select([Person.party_id,
                Person.first_name.label('first_name'),
                Person.last_name.label('last_name'),
                Person.birthdate.label('birthdate'),
                Person.social_security_number.label('social_security_number'),
                Person.passport_number.label('passport_number'),
                func.sum( VisitorReport.visitors ).label('visitors'),],
                whereclause = and_( Person.party_id == Movie.director_party_id,
                                    Movie.id == VisitorReport.movie_id),
                group_by = [ Person.party_id,
                             Person.first_name,
                             Person.last_name,
                             Person.birthdate,
                             Person.social_security_number,
                             Person.passport_number, ] )

    s=s.alias('visitors_per_director')

    mapper( VisitorsPerDirector, s, always_refresh=True )
```

Put all this in a file called view.py

Put into action

Then make sure the plain old Python object is mapped to the query, just after the Elixir model has been setup, by modifying the setup_model function in settings.py:

```
    def setup_model():
        from sqlalchemy.orm import configure_mappers
        from camelot.core.sql import metadata
        metadata.bind = settings.ENGINE()
        import camelot.model.party
        import camelot.model.authentication
        import camelot.model.i18n
        import camelot.model.fixture
        import camelot.model.memento
        import camelot.model.batch_job
        import camelot_example.model
        #
        # create the tables for all models, configure mappers first, to make
        # sure all deferred properties have been handled, as those could
        # create tables or columns
        #
        configure_mappers()
        metadata.create_all()
        from camelot.model.authentication import update_last_login
        #update_last_login()
        #
        # Load sample data with the fixure mechanism
        #
        from camelot_example.fixtures import load_movie_fixtures
        load_movie_fixtures()
        #
        # setup the views
        #
        from camelot_example.view import setup_views
        setup_views()
```

And add the plain old Python object to a section in the **ApplicationAdmin**:

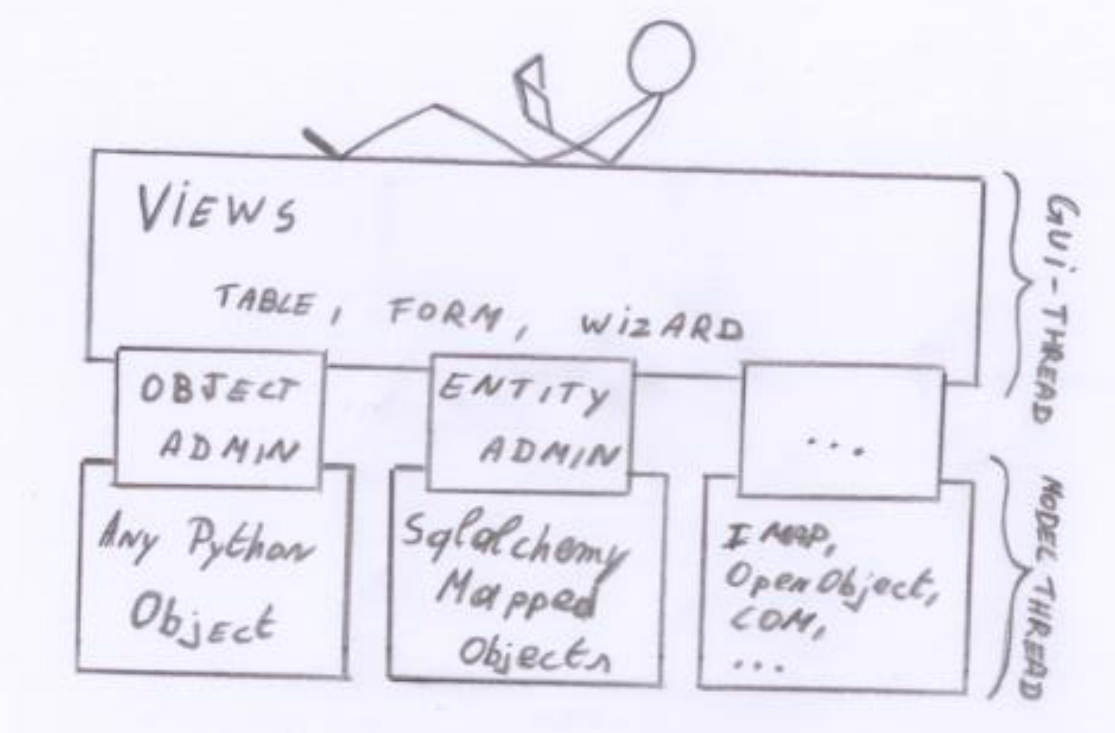
```
    def get_sections(self):

        from camelot.model.batch_job import BatchJob
        from camelot.model.memento import Memento
        from camelot.model.party import ( Person, Organization,
                                          PartyCategory )
        from camelot.model.i18n import Translation
        from camelot.model.batch_job import BatchJob, BatchJobType

        from camelot_example.model import Movie, Tag, VisitorReport
        from camelot_example.view import VisitorsPerDirector
# begin import action
        from camelot_example.importer import ImportCovers
# end import action

        return [
```

```
# begin section with action
                Section( _('Movies'),
                         self,
                         Icon('tango/22x22/mimetypes/x-office-presentation.png'),
                         items = [ Movie,
                                   Tag,
                                   VisitorReport,
#                                   VisitorsPerDirector,
                                   ImportCovers() ]),
# end section with action
                Section( _('Relation'),
                         self,
                         Icon('tango/22x22/apps/system-users.png'),
                         items = [ Person,
                                   Organization,
                                   PartyCategory ]),
                Section( _('Configuration'),
                         self,
                         Icon('tango/22x22/categories/preferences-system.png'),
                         items = [ Memento,
                                   Translation,
                                   BatchJobType,
                                   BatchJob
                                   ])
                ]
```

doc/../_static/entityviews/table_view_visitorsperdirector.png

3.3 Admin classes

The Admin classes are the classes that specify how objects should be visualized, they define the look, feel and behaviour of the Application. Most of the behaviour of the Admin classes can be tuned by changing their class attributes. This makes it easy to subclass a default Admin class and tune it to your needs.



3.3.1 ObjectAdmin

Camelot is able to visualize any Python object, through the use of the `camelot.admin.object_admin.ObjectAdmin` class. However, subclasses exist that use introspection to facilitate the visualisation.

Each class that is visualized within Camelot has an associated Admin class which specifies how the object or a list of objects should be visualized.

Usually the Admin class is bound to the model class by defining it as an inner class of the model class:

```
class Options(object):
    """A python object in which we store the change in rating
    """

    def __init__(self):
        self.only_selected = True
        self.change = 1

    # Since Options is a plain old python object, we cannot
    # use an EntityAdmin, and should use the ObjectAdmin
    class Admin( ObjectAdmin ):
        verbose_name = _('Change rating options')
        form_display = ['change', 'only_selected']
        form_size = (100, 100)
        # Since there is no introspection, the delegate should
        # be specified explicitely, and set to editable
        field_attributes = {'only_selected':{'delegate':delegates.BoolDelegate,
                                             'editable':True},
                            'change':{'delegate':delegates.IntegerDelegate,
```

```
                                        'editable':True},
                                }

# begin change rating action definition
```

Most of the behaviour of the Admin class can be customized by changing the class attributes like *verbose_name*, *list_display* and *form_display*.

Other *Admin* classes can inherit *ObjectAdmin* if they want to provide additional functionallity, like introspection to set default field attributes.

3.3.2 EntityAdmin

The `camelot.admin.entity_admin.EntityAdmin` class is a subclass of *ObjectAdmin* that can be used to visualize objects mapped to a database using SQLAlchemy.

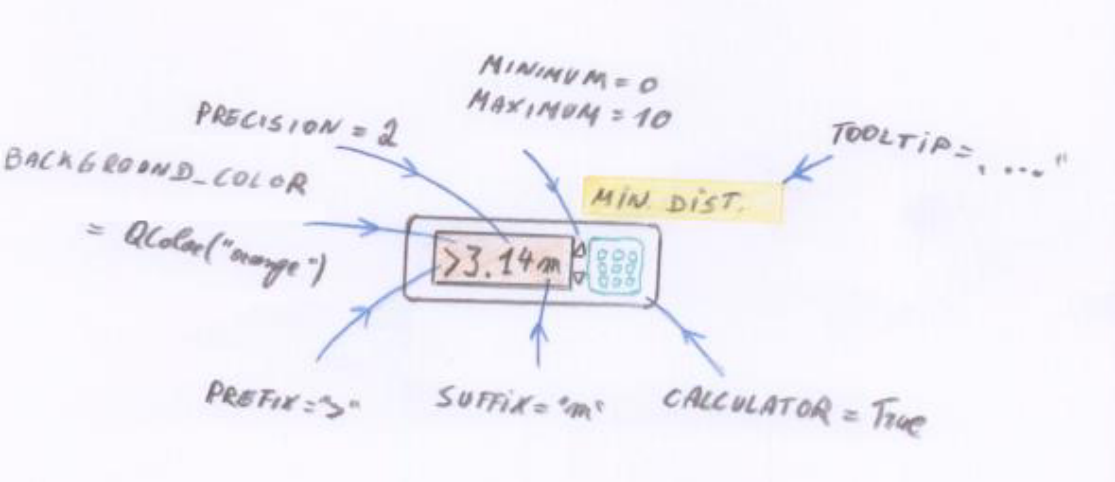
The *EntityAdmin* uses introspection of the model to guess the default field attributes. This makes the definition of an *Admin* class less verbose.

```
class Tag(Entity):

    __tablename__ = 'tags'

    name = Column( sqlalchemy.types.Unicode(60), nullable = False )
    movies = ManyToMany( 'Movie',
                         tablename = 'tags_movies__movies_tags',
                         local_colname = 'movies_id',
                         remote_colname = 'tags_id' )

    def __unicode__( self ):
        return self.name

    class Admin( EntityAdmin ):
        form_size = (400,200)
        list_display = ['name']

# begin visitor report definition
```

The `camelot.admin.entity_admin.EntityAdmin` provides some additonal attributes on top of those provided by `camelot.admin.object_admin.ObjectAdmin`, such as *list_filter* and *list_search*

3.3.3 Others

Field Attributes



Field attributes are the most convenient way to customize an application, they can be specified through the *field_attributes* dictionary of an *Admin* class :

```
class VisitorReport(Entity):

    __tablename__ = 'visitor_report'

    date = Column( sqlalchemy.types.Date,
                   nullable = False,
                   default = datetime.date.today )
    visitors = Column( sqlalchemy.types.Integer,
                       nullable = False,
                       default = 0 )
    movie = ManyToOne( 'Movie', required = True )
# end visitor report definition

    class Admin(EntityAdmin):
        verbose_name = _('Visitor Report')
        list_display = ['movie', 'date', 'visitors']
        field_attributes = {'visitors':{'minimum':0}}
```

Each combination of a delegate and an editor used to handle a field supports a different set of field attributes. To know which field attribute is supported by which editor or delegate, have a look at the *Delegates* documentation.

Static Field Attributes

Static field attributes should be the same for every row in the same column, as such they should be specified as constant in the field attributes dictionary.

Dynamic Field Attributes

Some field attributes, like background_color, can be dynamic. This means they can be specified as a function in the field attributes dictionary.

This function should take as its single argument the object on which the field attribute applies, as can be seen in the *background color example*

These are the field attributes that can be dynamic:

Overview of the field attributes

address_validator A function that verifies if a virtual address is valid, and eventually corrects it. The default implementation can is `camelot.view.controls.editors.virtualaddresseditor.default_address_validator()`

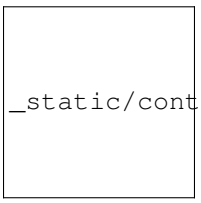
This function will be called while the user is editing the address, therefor it should take very little time to do the validation. If the address is invalid, this will be shown to the user, but it will not block the input of the address.

calculator `True` or `False` Indicates whether a calculator should be available when editing this field.

create_inline used in a one to many relation, if `False`, then a new entity will be created within a new window, if `True`, it will be created as a new line in the table.

column_width An integer forcing the column width of a field in a table view. The use of this field attribute is not recommended, since in most cases Camelot will figure out how wide a column should be. The use of *minimal_column_width* is advised to make sure a column has a certain width. But the *column_width* field attribute can be used to shrink the column width to arbitrary sizes, even if this might make the header unreadeable.

```
field_attributes = { 'first_name':{'column_width':8},
                     'suffix':{'column_width':8},}
```



directory `True` or `False` indicates if the file editor should point to a directory instead of a file. By default it points to a file.

editable `True` or `False`

Indicates whether the user can edit the field.

field_name This is the object name of the `QtGui.QWidget` that will be used as an editor for this field.

file_filter When the user is able to select a file or filename, use this filter to limit the available files.

length The maximum number of characters that can be entered in a text field.

minimum The minimum allowed value for `Integer` and `Float` delegates or their related delegates like the Star delegate.

maximum The maximum allowed value for `Integer` and `Float` delegates or their related delegates like the Star delegate.

precision The numerical precision that will be used to display `Float` values, this is unrelated to the precision in which they are stored.

choices A function taking as a single argument the object to which the field belongs. The function returns a list of tuples containing for each possible choice the value to be stored on the model and the value displayed to the user.

The use of `choices` forces the use of the ComboBox delegate:

```
field_attributes = {'state':{'choices':lambda o:[(1, 'Active'),
                                                 (2, 'Passive')]}}
```

minimal_column_width An integer specifying the minimal column width when this field is displayed in a table view. The width is expressed as the number of characters that should fit in the column:

```
field_attributes = {'name':{'minimal_column_width':50}}
```

will make the column wide enough to display at least 50 characters. The user will still be able to reduce the column size manually.

prefix String to display before a number

remove_original `True` or `False`

Set to `True` when a file should be deleted after it has been transfered to the storage.

single_step The size of a single step when the up and down arrows are used in on a float or an integer field.

suffix String to display after a number

tooltip A function taking as a single argument the object to which the field belongs. The function should return a string that will be used as a tooltip. The string may contain html markup.

```
from camelot.admin.object_admin import ObjectAdmin
from camelot.view.controls import delegates

def dynamic_tooltip_x(coordinate):
  return u'The <b>x</b> value of the coordinate, now set to %s'%(coordinate.x)

def dynamic_tooltip_y(coordinate):
  return u'The <b>y</b> value of the coordinate, now set to %s'%(coordinate.y)

class Coordinate(object):

  def __init__(self):
    self.id = 1
    self.x = 0.0
    self.y = 0.0

  class Admin(ObjectAdmin):
```

```
    form_display = ['x', 'y',]
    field_attributes = dict(x=dict(delegate=delegates.FloatDelegate,
                                   tooltip=dynamic_tooltip_x),
                            y=dict(delegate=delegates.FloatDelegate,
                                   tooltip=dynamic_tooltip_y),
                            )
    form_size = (100,100)
```

translate_content `True` or `False`

Wether the content of a field should be translated before displaying it. This only works for displaying content, not while editing it.

background_color A function taking as a single argument the object to which the field belongs. The function should return None if the default background should be used, or a QColor to be used as the background.

```
"""This Admin class turns the background of a Person's first
name pink if its first name doesn't start with a capital"""

from PyQt4.QtGui import QColor

from camelot.model.party import Person

def first_name_background_color(person):
    import string
    if person.first_name:
        if person.first_name[0] not in string.uppercase:
            return QColor('pink')

class Admin(Person.Admin):
    field_attributes = {'first_name':{'background_color':first_name_background_color}}
```

name The name of the field used, this defaults to the name of the attribute

target In case of relation fields, specifies the class that is at the other end of the relation. Defaults to the one found by introspection. This can be used to let a many2one editor always point to a subclass of the one found by introspection.

admin In case of relation fields, specifies the admin class that is to be used to visualize the other end of the relation. Defaults to the default admin class of the target class. This can be used to make the table view within a one2many widget look different from the default table view for the same object.

address_type Should be None or one of the Virtual Address Types, like 'phone' or 'email'. When specified, it indicates that a VirtualAddressEditor should only accept addresses of the specified type.

Customizing multiple field attributes

When multiple field attributes need to be customized, specifying the *field_attributes* dictionary can become inefficient.

Several methods of the `camelot.admin.object_admin.ObjectAdmin` class can be overwritten to take care of this.

Instead of filling the *field_attributes* dictionary manually, the **:method:'camelot.admin.object_admin.ObjectAdmin.get_field_attribu** method can be overwritten :

When multiple dynamic field attributes need to execute the same logic to determine their value, it can be more efficient to overwrite the method **:method:'camelot.admin.object_admin.ObjectAdmin.get_dynamic_field_attributes'** and execute the logic once there and set the value for all dynamic field attributes at once.

The complement of *get_dynamic_field_attributes* is **:method:'camelot.admin.object_admin.ObjectAdmin.get_static_field_attribute**

Validators

Before an object is written to the database it needs to be validated, and the user needs to be informed in case the object is not valid.

By default Camelot does some introspection on the model to check the validity of an object, to make sure it will be able to write the object to the database.

But this might not be enough. If more validation is needed, a custom Validator class can be defined. The default `camelot.admin.validator.entity_validator.EntityValidator` can be subclassed to create a custom validator. The new class should then be bound to the Admin class :

```
from camelot.admin.validator.entity_validator import EntityValidator
from camelot.admin.entity_admin import EntityAdmin

class PersonValidator(EntityValidator):

    def objectValidity(self, entity_instance):
        messages = super(PersonValidator,self).objectValidity(entity_instance)
        if (not entity_instance.first_name) or (len(entity_instance.first_name) < 3):
            messages.append("A person's first name should be at least 2 characters long")
        return messages

class Admin(EntityAdmin):
    verbose_name = 'Person'
    list_display = ['first_name', 'last_name']
    validator = PersonValidator
```

Its most important method is objectValidity, which takes an object as argument and should return a list of strings explaining why the object is invalid. These strings will then be presented to the user.

Notice that this method will always get called outside of the GUI thread, so the call will never block the GUI.

When the user tries to leave a form in an invalid state, a platform dependent dialog box will appear.

3.4 Customizing the Application

The **ApplicationAdmin** controls how the application behaves, it determines the sections in the left pane, the availability of help, the about box, the menu structure, etc.

3.4.1 The Application Admin

Each Camelot application should subclass `camelot.admin.application_admin.ApplicationAdmin` and overwrite some of its methods.

The look of the main window

Most of these methods are based on the concept of *Actions*.

- `camelot.admin.application_admin.ApplicationAdmin.get_sections()`
- `camelot.admin.application_admin.ApplicationAdmin.get_actions()`
- `camelot.admin.application_admin.ApplicationAdmin.get_toolbar_actions()`
- `camelot.admin.application_admin.ApplicationAdmin.get_main_menu()`

Interaction with the Operating System

- `camelot.admin.application_admin.ApplicationAdmin.get_organization_name()`
- `camelot.admin.application_admin.ApplicationAdmin.get_organization_domain()`
- `camelot.admin.application_admin.ApplicationAdmin.get_name()`
- `camelot.admin.application_admin.ApplicationAdmin.get_version()`

The look of the application

- `camelot.admin.application_admin.ApplicationAdmin.get_splashscreen()`
- `camelot.admin.application_admin.ApplicationAdmin.get_stylesheet()`
- `camelot.admin.application_admin.ApplicationAdmin.get_translator()`
- `camelot.admin.application_admin.ApplicationAdmin.get_icon()`

The content of the help menu

- `camelot.admin.application_admin.ApplicationAdmin.get_about()`
- `camelot.admin.application_admin.ApplicationAdmin.get_help_url()`

Default behavior of the application

- camelot.admin.application_admin.ApplicationAdmin.get_related_admin()

The look of the form views

- camelot.admin.application_admin.ApplicationAdmin.get_related_toolbar_actions()
- camelot.admin.application_admin.ApplicationAdmin.get_form_actions()
- camelot.admin.application_admin.ApplicationAdmin.get_form_toolbar_actions()

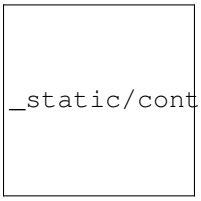
Example

```
class MyApplicationAdmin(ApplicationAdmin):

    name = 'Camelot Video Store'

# begin sections
    def get_sections(self):

        from camelot.model.batch_job import BatchJob
        from camelot.model.memento import Memento
        from camelot.model.party import ( Person, Organization,
                                          PartyCategory )
        from camelot.model.i18n import Translation
        from camelot.model.batch_job import BatchJob, BatchJobType

        from camelot_example.model import Movie, Tag, VisitorReport
        from camelot_example.view import VisitorsPerDirector
# begin import action
        from camelot_example.importer import ImportCovers
# end import action

        return [
# begin section with action
                Section( _('Movies'),
                         self,
                         Icon('tango/22x22/mimetypes/x-office-presentation.png'),
                         items = [ Movie,
                                   Tag,
                                   VisitorReport,
#                                    VisitorsPerDirector,
                                   ImportCovers() ]),
# end section with action
                Section( _('Relation'),
                         self,
                         Icon('tango/22x22/apps/system-users.png'),
                         items = [ Person,
                                   Organization,
                                   PartyCategory ]),
                Section( _('Configuration'),
                         self,
                         Icon('tango/22x22/categories/preferences-system.png'),
                         items = [ Memento,
                                   Translation,
                                   BatchJobType,
```

```
                                    BatchJob
                                    ])
                ]
# end sections

# begin actions
    def get_actions(self):
        from camelot.admin.action import OpenNewView
        from camelot_example.model import Movie

        new_movie_action = OpenNewView( self.get_related_admin(Movie) )
        new_movie_action.icon = Icon('tango/22x22/mimetypes/x-office-presentation.png')

        return [new_movie_action]
# end actions
```

3.4.2 Example of a reduced application

By reimplementing the default `get_sections()`, `get_main_menu()` and `get_toolbar_actions()`, it is possible to create a completely differently looking Camelot application.




```
    def get_toolbar_actions( self, toolbar_area ):
        from PyQt4.QtCore import Qt
        from camelot.model.party import Person
        from camelot.admin.action import application_action, list_action
        from model import Movie

        movies_action = application_action.OpenTableView( self.get_related_admin( Movie ) )
        movies_action.icon = Icon('tango/22x22/mimetypes/x-office-presentation.png')
        persons_action = application_action.OpenTableView( self.get_related_admin( Person ) )
        persons_action.icon = Icon('tango/22x22/apps/system-users.png')

        if toolbar_area == Qt.LeftToolBarArea:
            return [ movies_action,
                     persons_action,
                     list_action.OpenNewView(),
                     list_action.OpenFormView(),
                     list_action.DeleteSelection(),
                     application_action.Exit(),]

    def get_actions( self ):
        return []

    def get_sections( self ):
        return None

    def get_main_menu( self ):
        return None
```

```
    def get_stylesheet(self):
        from camelot.view import art
        return art.read('stylesheet/black.qss')
```

3.5 Creating Forms

This section describes how to place fields on forms and applying various layouts. It also covers how to customize forms to your specific needs. As with everything in Camelot, the goal of the framework is that you can create 80% of your forms with minimal effort, while the framework should allow you to really customize the other 20% of your forms.

3.5.1 Form

A form is a collection of fields organized within a layout. Each field is represented by its editor.

Usually forms are defined by specifying the *form_display* attribute of an Admin class :

```
from sqlalchemy.schema import Column
from sqlalchemy.types import Unicode, Date
from camelot.admin.entity_admin import EntityAdmin
from camelot.core.orm import Entity
from camelot.view import forms

class Movie( Entity ):
    title = Column( Unicode(60), nullable=False )
    short_description = Column( Unicode(512) )
    releasedate = Column( Date )

    class Admin(EntityAdmin):
        form_display = forms.Form( ['title', 'short_description', 'releasedate'] )
```

The *form_display* attribute should either be a list of fields to display or an instance of camelot.view.forms.Form or its subclasses.

Forms can be nested into each other :

```
from camelot.admin.entity_admin import EntityAdmin
from camelot.view import forms
from camelot.core.utils import ugettext_lazy as _

class Admin(EntityAdmin):
    verbose_name = _('person')
    verbose_name_plural = _('persons')
    list_display = ['first_name', 'last_name', ]
    form_display = forms.TabForm([('Basic', forms.Form(['first_name', 'last_name', 'contact_mechanis
                                   ('Official', forms.Form(['birthdate', 'social_security_number', 'pa
                                                            'passport_expiry_date','addresses',])), ])
```

3.5.2 Inheritance and Forms

Just as Entities support inheritance, forms support inheritance as well. This avoids duplication of effort when designing and maintaining forms. Each of the Form subclasses has a set of methods to modify its content. In the example below a new tab is added to the form defined in the previous section.

```
from copy import deepcopy

from camelot.view import forms
from nested_form import Admin

class InheritedAdmin(Admin):
    form_display = deepcopy(Admin.form_display)
    form_display.add_tab('Work', forms.Form(['employers', 'directed_organizations', 'shares']))
```

3.5.3 Putting notes on forms

A note on a form is nothing more than a property with the NoteDelegate as its delegate and where the widget is inside a WidgetOnlyForm.

In the case of a Person, we display a note if another person with the same name already exists :

```
    def note(self):
        for person in self.__class__.query.filter_by(first_name=self.first_name, last_name=self.last_
            if person != self:
                return _('A person with the same name already exists')
```

3.5.4 Available Form Subclasses

The `camelot.view.forms.Form` class has several subclasses that can be used to create various layouts. Those can be found in the `camelot.view.forms` module. Each subclass maps to a Qt Layout class.

3.5.5 Customizing Forms

Several options exist for completely customizing the forms of an application.

Layout

When the desired layout cannot be achieved with Camelot's form classes, a custom `camelot.view.forms.Form` subclass can be made to layout the widgets.

When subclassing the *Form* class, it's *render* method should be reimplemented to put the labels and the editors in a custom layout. The *render* method will be called by Camelot each time it needs the form. It should thus return a `QtGui.QWidget` to be used as the needed form.

The *render* method its first argument is the factory class `camelot.view.controls.formview.FormEditors`, through which editors and labels can be constructed. The editor widgets are bound to the data model.

```
from PyQt4 import QtGui

from camelot.view import forms
from camelot.admin.entity_admin import EntityAdmin

class CustomForm( forms.Form ):

    def __init__(self):
        super( CustomForm, self ).__init__(['first_name', 'last_name'])

    def render( self, editor_factory, parent = None, nomargins = False ):
        widget = QtGui.QWidget( parent )
        layout = QtGui.QFormLayout()
        layout.addRow( QtGui.QLabel('Please fill in the complete name :', widget ) )
        for field_name in self.get_fields():
            field_editor = editor_factory.create_editor( field_name, widget )
            field_label = editor_factory.create_label( field_name, field_editor, widget )
            layout.addRow( field_label, field_editor )
        widget.setLayout( layout )
        widget.setBackgroundRole( QtGui.QPalette.ToolTipBase )
        widget.setAutoFillBackground( True )
        return widget

class Admin(EntityAdmin):
    list_display = ['first_name', 'last_name']
    form_display = CustomForm()
    form_size = (300,100)
```

The form defined above puts the widgets into a `QtGui.QFormLayout` using a different background color, and adds some instructions for the user :

doc/../_static/form/custom_layout.png

Editors

The editor of a specific field can be changed, by specifying an alternative `QtGui.QItemDelegate` for that field, using the *delegate* field attributes, see *Specifying delegates*.

Tooltips

Each field on the form can be given a dynamic tooltip, using the *tooltip* field attribute, see *tooltip*.

Buttons

Buttons bound to a specific action can be put on a form, using the *form_actions* attribute, attribute of the Admin class : *Form Actions*.

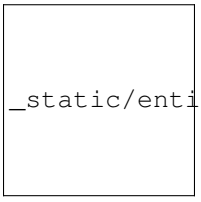
Validation

Validation is done at the object level. Before a form is closed validation of the bound object takes place, an invalid object will prevent closing the form. A custom validator can be defined : *Validators*

3.6 Actions

3.6.1 Introduction

Besides displaying and editing data, every application needs the functions to manipulate data or create reports. In Camelot this is done through actions. Actions can appear as buttons on the side of a form or a table, as icons in a toolbar or as icons in the home workspace.



Every Action is build up with a set of Action Steps. An Action Step is a reusable part of an Action, such as for example, ask the user to select a file. Camelot comes with a set of standard Actions and Action Steps that are easily extended to manipulate data or create reports.

When defining Actions, a clear distinction should be made between things happening in the model thread (the manipulation or querying of data), and things happening in the gui thread (pop up windows or reports). The *The Two Threads* section gives more detail on this.

3.6.2 Summary

In general, actions are defined by subclassing the standard Camelot `camelot.admin.action.Action` class

```
from camelot.admin.action import Action
from camelot.view.action_steps import PrintHtml
from camelot.core.utils import ugettext_lazy as _
from camelot.view.art import Icon
```

```
class PrintReport( Action ):

    verbose_name = _('Print Report')
    icon = Icon('tango/16x16/actions/document-print.png')
    tooltip = _('Print a report with all the movies')

    def model_run( self, model_context ):
        yield PrintHtml( 'Hello World' )
```

Each action has two methods, `gui_run()` and `model_run()`, one of them should be reimplemented in the subclass to either run the action in the gui thread or to run the action in the model thread. The default `Action.gui_run()` behavior is to pop-up a `ProgressDialog` dialog and start the `model_run()` method in the model thread.

`model_run()` in itself is a generator, that can yield `ActionStep` objects back to the gui, such as a `PrintHtml`.

The action objects can than be used a an element of the actions list returned by the `ApplicationAdmin.get_actions()` method:

```
    def get_actions(self):
        from camelot.admin.action import OpenNewView
        from camelot_example.model import Movie

        new_movie_action = OpenNewView( self.get_related_admin(Movie) )
        new_movie_action.icon = Icon('tango/22x22/mimetypes/x-office-presentation.png')

        return [new_movie_action]
```

or be used in the `ObjectAdmin.list_actions` or `ObjectAdmin.form_actions` attributes.

The *Add an import wizard to an application* tutorial has a complete example of creating and using and action.

3.6.3 What can happen inside `model_run()`

`yield` events to the GUI

Actions need to be able to send their results back to the user, or ask the user for additional information. This is done with the `yield` statement.

Through `yield`, an Action Step is send to the GUI thread, where it creates user interaction, and sends it result back to the 'model_thread'. The model_thread will be blocked while the action in the GUI thread takes place, eg

```
yield PrintHtml( 'Hello World' )
```

Will pop up a print preview dialog in the GUI, and the model_run method will only continue when this dialog is closed.

Events that can be yielded to the GUI should be of type `camelot.admin.action.base.ActionStep`. Action steps are reusable parts of an action. Possible Action Steps that can be yielded to the GUI include:

- `camelot.view.action_steps.change_object.ChangeObject`
- `camelot.view.action_steps.change_object.ChangeObjects`
- `camelot.view.action_steps.print_preview.PrintChart`
- `camelot.view.action_steps.print_preview.PrintPreview`
- `camelot.view.action_steps.print_preview.PrintHtml`
- `camelot.view.action_steps.print_preview.PrintJinjaTemplate`
- `camelot.view.action_steps.open_file.OpenFile`

- camelot.view.action_steps.open_file.OpenStream
- camelot.view.action_steps.open_file.OpenJinjaTemplate
- camelot.view.action_steps.gui.CloseView
- camelot.view.action_steps.gui.MessageBox
- camelot.view.action_steps.gui.Refresh
- camelot.view.action_steps.gui.OpenFormView
- camelot.view.action_steps.gui.ShowPixmap
- camelot.view.action_steps.gui.ShowChart
- camelot.view.action_steps.select_file.SelectFile
- camelot.view.action_steps.select_object.SelectObject

keep the user informed about progress

An camelot.view.action_steps.update_progress.UpdateProgress object can be yielded, to update the state of the progress dialog:

This should be done regulary to keep the user informed about the progres of the action:

```
movie_count = Movie.query.count()

report = '<table>'
for i, movie in enumerate( Movie.query.all() ):
    report += '<tr><td>%s</td></tr>'%(movie.name)
    yield UpdateProgress( i, movie_count )
report += '</table>'

yield PrintHtml( report )
```

Should the user have pressed the *Cancel* button in the progress dialog, the next yield of an UpdateProgress object will raise a camelot.core.exception.CancelRequest.

manipulation of the model

The most important purpose of an action is to query or manipulate the model, all such things can be done in the model_run() method, such as executing queries, manipulating files, etc.

Whenever a part of the model has been changed, it might be needed to inform the GUI about this, so that it can update itself, the easy way of doing so is by yielding an instance of camelot.view.action_steps.orm.FlushSession such as:

```
movie.rating = 5
yield FlushSession( model_context.session )
```

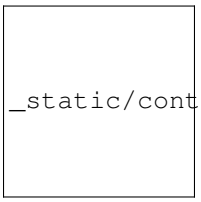
This will flush the session to the database, and at the same time update the GUI so that the flushed changes are shown to the user by updating the visualisation of the changed movie on every screen in the application that displays this object. Alternative updates that can be generated are :

- camelot.view.action_steps.orm.UpdateObject, if one wants to inform the GUI an object has been updated.
- camelot.view.action_steps.orm.DeleteObject, if one wants to inform the GUI an object is going to be deleted.

- `camelot.view.action_steps.orm.CreateObject`, if one wants to inform the GUI an object has been created.

raise exceptions

When an action fails, a normal Python `Exception` can be raised, which will pop-up an exception dialog to the user that displays a stack trace of the exception. In case no stack trace should be shown to the user, a `camelot.core.exception.UserException` should be raised. This will popup a friendly dialog :



When the `model_run()` method raises a `camelot.core.exception.CancelRequest`, a `GeneratorExit` or a `StopIteration` exception, these are ignored and nothing will be shown to the user.

handle exceptions

In case an unexpected event occurs in the GUI, a `yield` statement will raise a `camelot.core.exception.GuiException`. This exception will propagate through the action an will be ignored unless handled by the developer.

request information from the user

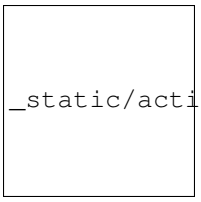
The pop-up of a dialog that presents the user with a number of options can be triggered from within the `model_run()` method. This happens by transferring an **options** object back and forth between the **model_thread** and the **gui_thread**. To transfer such an object, this object first needs to be defined:

```
class Options( object ):

    def __init__(self):
        self.earliest_releasedate = datetime.date(2000, 1, 1)
        self.latest_releasedate = datetime.date.today()

    class Admin( ObjectAdmin ):
        form_display = [ 'earliest_releasedate', 'latest_releasedate' ]
        field_attributes = { 'earliest_releasedate':{'delegate':delegates.DateDelegate},
                             'latest_releasedate':{'delegate':delegates.DateDelegate}, }
```

Than a `camelot.view.action_steps.change_object.ChangeObject` action step can be `yield` to present the options to the user and get the filled in values back :

```
        from PyQt4 import QtGui
        from camelot.view import action_steps
        options = NewProjectOptions()
        yield action_steps.UpdateProgress( text = 'Request information' )
        yield action_steps.ChangeObject( options )
```

Will show a dialog to modify the object:



When the user presses *Cancel* button of the dialog, the `yield` statement will raise a `camelot.core.exception.CancelRequest`.

Other ways of requesting information are :

- `camelot.view.action_steps.select_file.SelectFile`, to request to select an existing file to process or a new file to save information.

Issue SQLAlchemy statements

Camelot itself only manipulates the database through objects of the ORM for the sake of make no difference between objects mapped to the database and plain old python objects. But for performance reasons, it is often desired to do manipulations directly through SQLAlchemy ORM or Core queries :

```
model_context.session.query( BatchJobType ).update( values = {'name':'accounting audit'},
                                                     synchronize_session = 'evaluate' )
```

3.6.4 States and Modes

States

The widget that is used to trigger an action can be in different states. A `camelot.admin.action.base.State` object is returned by the `camelot.admin.action.base.Action.get_state` method. Subclasses of Action can reimplement this method to change the State of an action button.

This allows to hide or disable the action button, depending on the objects selected or the current object being displayed.

Modes

An action widget can be triggered in different modes, for example a print button can be triggered as *Print* or *Export to PDF*. The different modes of an action are specified as a list of `camelot.admin.action.base.Mode` objects.

To change the modes of an Action, either specify the `modes` attribute of an `Action` or specify the `modes` attribute of the `State` returned by the `Action.get_state()` method.

3.6.5 Action Context

Depending on where an action was triggered, a different context will be available during its execution in `camelot.admin.action.base.Action.gui_run()` and `camelot.admin.action.base.Action.model_run()`.

The minimal context available in the *GUI thread* when `gui_run()` is called :

While the minimal contact available in the *Model thread* when `model_run()` is called :

Application Actions

To enable Application Actions for a certain `ApplicationAdmin` overwrite its `ApplicationAdmin.get_actions()` method:

```
from camelot.admin.application_admin import ApplicationAdmin
from camelot.admin.action import Action

class GenerateReports( Action ):

    verbose_name = _('Generate Reports')

    def model_run( self, model_context):
        for i in range(10):
            yield UpdateProgress(i, 10)

class MyApplicationAdmin( ApplicationAdmin )

    def get_actions( self ):
        return [GenerateReports(),]
```

An action specified here will receive an `ApplicationActionGuiContext` object as the *gui_context* argument of the the `gui_run()` method, and a `ApplicationActionModelContext` object as the *model_context* argument of the `model_run()` method.

Form Actions

A form action has access to the object currently visible on the form.

```
class BurnToDisk( Action ):

    verbose_name = _('Burn to disk')

    def model_run( self, model_context ):
        yield action_steps.UpdateProgress( 0, 3, _('Formatting disk') )
        time.sleep( 0.7 )
        yield action_steps.UpdateProgress( 1, 3, _('Burning movie') )
        time.sleep( 0.7 )
        yield action_steps.UpdateProgress( 2, 3, _('Finishing') )
        time.sleep( 0.5 )
```

To enable Form Actions for a certain `ObjectAdmin` or `EntityAdmin`, specify the `form_actions` attribute.

```
        #
        # create a list of actions available for the user on the form view
        #
        form_actions = [BurnToDisk()]
```

_static/entityviews/new_view_movie.png

An action specified here will receive a `FormActionGuiContext` object as the *gui_context* argument of the `gui_run()` method, and a `FormActionModelContext` object as the *model_context* argument of the `model_run()` method.

List Actions

A list action has access to both all the rows displayed in the table (called the collection) and the rows selected by the user (called the selection) :

```
class ChangeRatingAction( Action ):
    """Action to print a list of movies"""

    verbose_name = _('Change Rating')

    def model_run( self, model_context ):
        #
        # the model_run generator method yields various ActionSteps
        #
        options = Options()
        yield ChangeObject( options )
        if options.only_selected:
            iterator = model_context.get_selection()
        else:
            iterator = model_context.get_collection()
        for movie in iterator:
            yield UpdateProgress( text = u'Change %s'%unicode( movie ) )
            movie.rating = min( 5, max( 0, (movie.rating or 0 ) + options.change ) )
        #
        # FlushSession will write the changes to the database and inform
        # the GUI
        #
        yield FlushSession( model_context.session )
```

To enable List Actions for a certain `ObjectAdmin` or `EntityAdmin`, specify the `list_actions` attribute:

```
        #
        # the action buttons that should be available in the list view
        #
        list_actions = [ChangeRatingAction()]
```

This will result in a button being displayed on the table view.

_static/entityviews/table_view_movie.png

An action specified here will receive a `ListActionGuiContext` object as the *gui_context* argument of th the `gui_run()` method, and a `ListActionModelContext` object as the *model_context* argument of the `model_run()` method.

Reusing List and Form actions

There is no need to define a different action subclass for form and list actions, as both their model_context have a **get_selection** method, a single action can be used both for the list and the form.

3.6.6 Available actions

Camelot has a set of available actions that combine the various `ActionStep` subclasses. Those actions can be used directly or as an inspiration to build new actions:

- `camelot.admin.action.application_action.OpenNewView`
- `camelot.admin.action.application_action.OpenTableView`
- `camelot.admin.action.application_action.ShowHelp`
- `camelot.admin.action.application_action.ShowAbout`
- `camelot.admin.action.application_action.Backup`
- `camelot.admin.action.application_action.Restore`
- `camelot.admin.action.application_action.Refresh`
- `camelot.admin.action.form_action.CloseForm`
- `camelot.admin.action.list_action.CallMethod`
- `camelot.admin.action.list_action.OpenFormView`
- `camelot.admin.action.list_action.OpenNewView`
- `camelot.admin.action.list_action.ToPreviousRow`
- `camelot.admin.action.list_action.ToNextRow`
- `camelot.admin.action.list_action.ToFirstRow`
- `camelot.admin.action.list_action.ToLastRow`
- `camelot.admin.action.list_action.ExportSpreadsheet`
- `camelot.admin.action.list_action.PrintPreview`
- `camelot.admin.action.list_action.SelectAll`
- `camelot.admin.action.list_action.ImportFromFile`
- `camelot.admin.action.list_action.ReplaceFieldContents`

3.6.7 Inspiration

- Implementing actions as generators was made possible with the language functions of **PEP 342**.
- The EuroPython talk of Erik Groeneveld inspired the use of these features. (http://ep2011.europython.eu/conference/talks/beyond-python-enhanched-generators)
- Action steps were introduced to be able to take advantage of the new language features of **PEP 380** in Python 3.3

3.7 Documents and Reports

3.7.1 Generate documents

Generating reports and documents is an important part of any application. Python and Qt provide various ways to generate documents. Each of them with its own advantages and disadvantages.

Method	Advantages	Disadvantages
PDF documents through reportlab	• Perfect control over layout • Excellent for mass creation of documents	• Relatively steep learning curve • User cannot edit document
HTML	• Easy to get started • Print preview within Camelot • No dependencies	• Not much layout control • User cannot edit document
Docx Word documents	• User can edit document	• Proprietary format • Word processor needed

Camelot leaves all options open to the developer.

Please have a look at *Creating a Report with Camelot* to get started with generating documents.

Generating a document or report is nothing more than yielding the appropriate action step during the `model_run()` method of an `Action`.

Action steps usable for reporting are :

- `camelot.view.action_steps.print_preview.PrintPreview`
- `camelot.view.action_steps.print_preview.PrintHtml`
- `camelot.view.action_steps.print_preview.PrintJinjaTemplate`
- `camelot.view.action_steps.open_file.OpenFile`
- `camelot.view.action_steps.open_file.OpenStream`
- `camelot.view.action_steps.open_file.OpenJinjaTemplate`

3.7.2 HTML based documents

```
class MovieSummary( Action ):

    verbose_name = _('Summary')

    def model_run(self, model_context):
        from camelot.view.action_steps import PrintHtml
        movie = model_context.get_object()
        yield PrintHtml( "<h1>This will become the movie report of %s!</h1>" % movie.title )
```

The supported html subset is documented here :

http://doc.qt.nokia.com/stable/richtext-html-subset.html

Alternative rendering

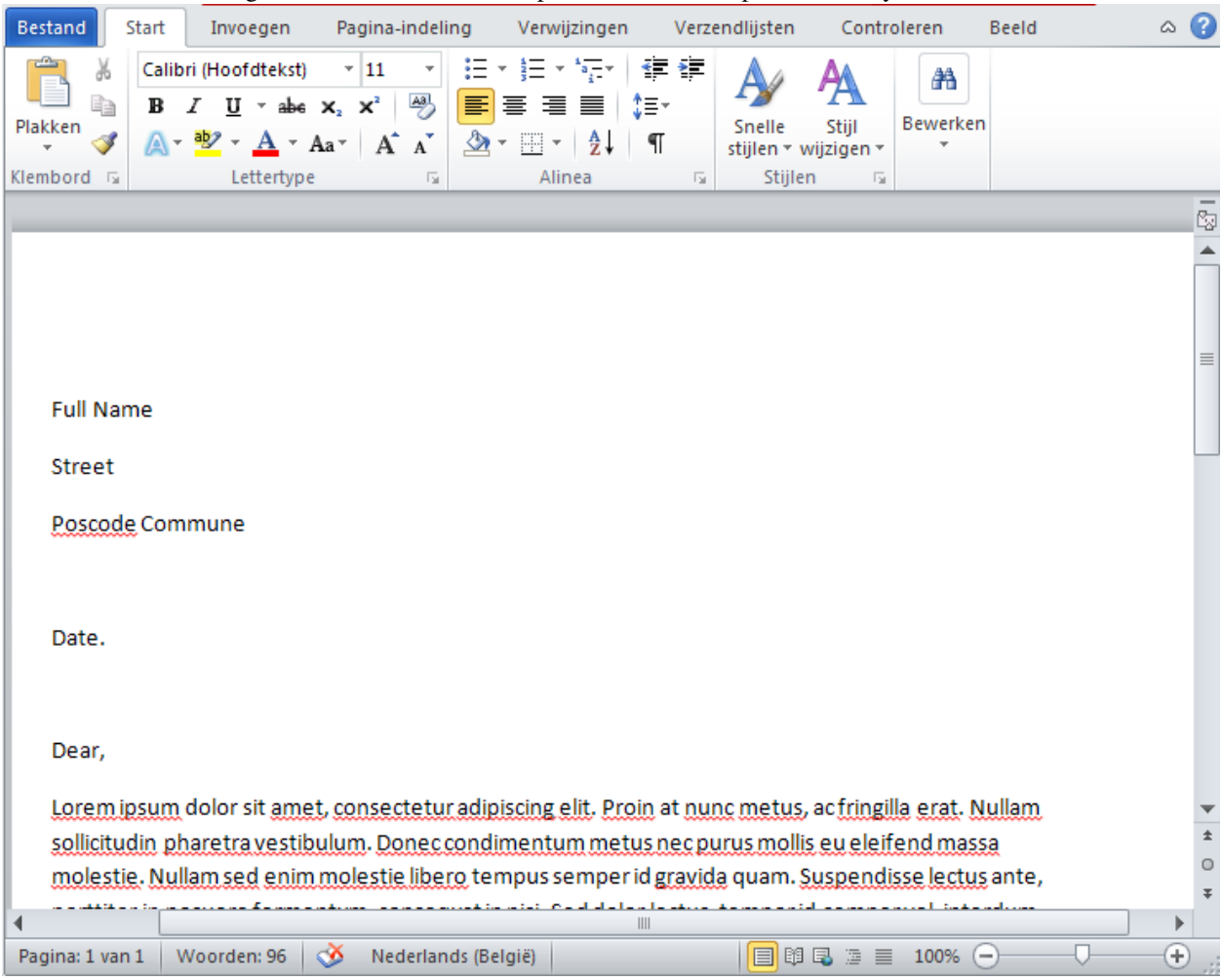
Instead of `QtGui.QTextDocument` another html renderer such as `QtWebKit.QWebView` can be used in combination with the `camelot.view.action_steps.print_preview.PrintPreview` action step. The `QWebView` class has complete support for html and css.

```
class WebkitPrint( Action ):

    def model_run( self, model_context ):
        from PyQt4.QtWebKit import QWebView
        from camelot.view.action_steps import PrintPreview

        movie = model_context.get_object()

        document = QWebView()
        document.setHtml( '<h2>%s</h2>' % movie.title )

        yield PrintPreview( document )
```

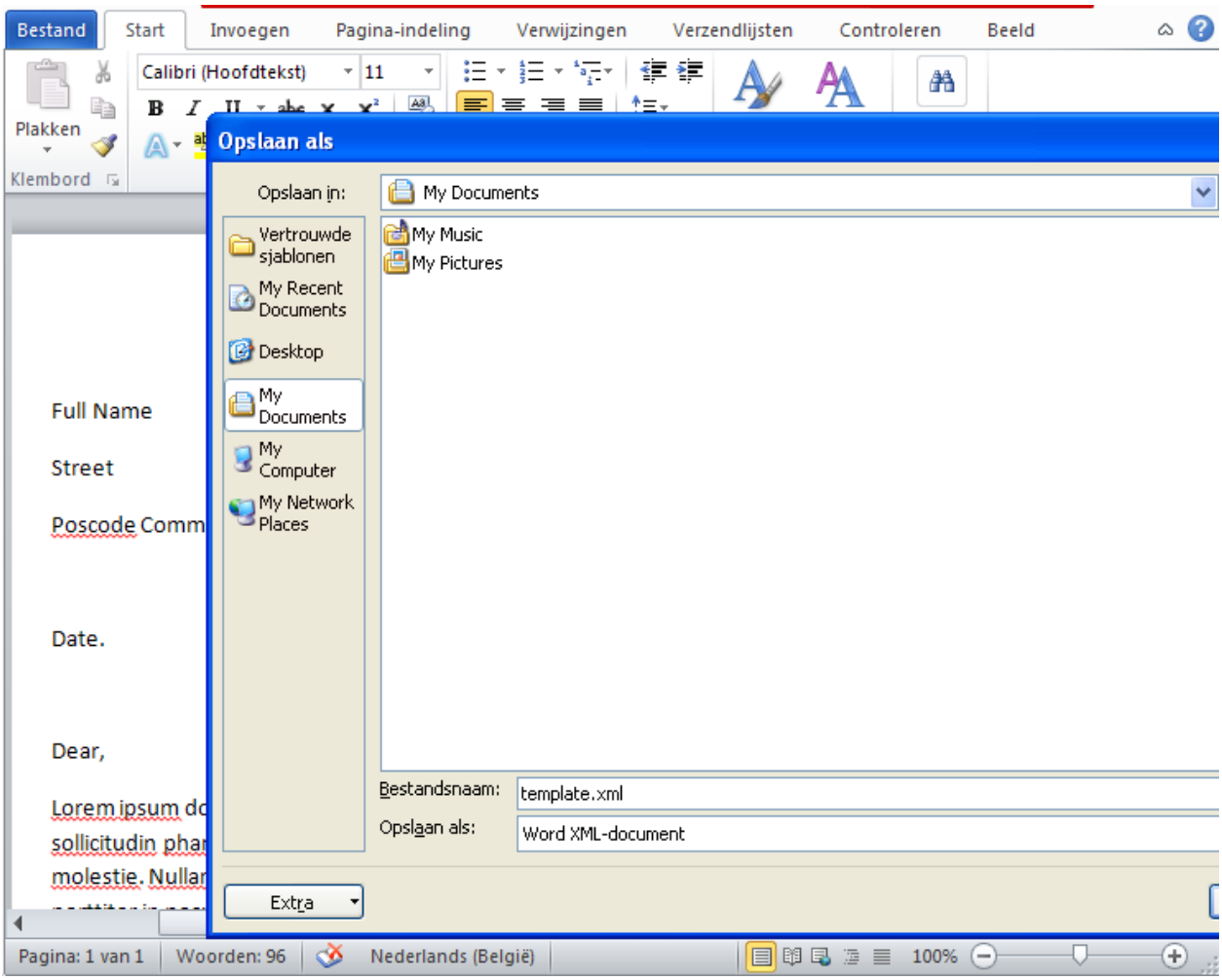
3.7.3 Docx based documents

Create a template document with MS Office

Create a document using MS Office and with some placeholder text on places where you want to insert data.



And save it as an xml file :



Clean the XML generated by MS Office

The XML file generated by MS Office can be cleaned using **xmllint**:

```
xmllint --format template.xml > template_clean.xml
```

Replace the placeholders

The template will be merged with the objects in the selection using jinja, where the object in the selection will be available as a variable named **obj** and the time of merging the document is available as **now**:

3.8 Delegates

Delegates are a cornerstone of the Qt model/delegate/view framework. A delegate is used to display and edit data from a *model*.

In the Camelot framework, every field of an *Entity* has an associated delegate that specifies how the field will be displayed and edited. When a new form or table is constructed, the delegates of all fields on the form or table will

construct *editors* for their fields and fill them with data from the model. When the data has been edited in the form, the delegates will take care of updating the model with the new data.

All Camelot delegates are subclasses of `QtGui.QAbstractItemDelegate`.

The Qt website provides detailed information the differenct classes involved in the model/delegate/view framework.

3.8.1 Specifying delegates

The use of a specific delegate can be forced by using the `delegate` field attribute. Suppose `rating` is a field of type `integer`, then it can be forced to be visualized as stars:

```
from camelot.view.controls import delegates

class Movie( Entity ):
    title = Column( Unicode(50) )
    rating = Column( Integer )

    class Admin( EntityAdmin ):
        list_display = ['title', 'rating']
        field_attributes = {'rating':{'delegate':delegates.StarDelegate}}
```

The above code will result in:

If no *delegate* field attribute is given, a default one will be taken depending on the sqlalchemy field type.

All available delegates can be found in `camelot.view.controls.delegates`

3.9 Charts

To enable charts, **Camelot** is closely integrate with Matplotlib, one of the very high quality Python charting packages.

Often creating a chart involves gathering a lot of data, this needs to happen inside the model, to free the GUI from such tasks. Once the data is gathered, it is put into a container, this container is then shipped to the gui thread, where the chart is put on the screen.

3.9.1 A simple plot

As shown in the example below, creating a simple plot involves two things :

1. Create a property that returns one of the chart containers, in this case the **PlotContainer** is used.

2. Specify the delegate to be used to visualize the property, this should be the **ChartDelegate**

```
from camelot.admin.object_admin import ObjectAdmin
from camelot.view.controls import delegates
from camelot.container.chartcontainer import PlotContainer

class Wave(object):

    def __init__(self):
        self.amplitude = 1
        self.phase = 0

    @property
    def chart(self):
        import math
        x_data = [x/100.0 for x in range(1, 700, 1)]
        y_data = [self.amplitude * math.sin(x - self.phase) for x in x_data]
        return PlotContainer( x_data, y_data )

    class Admin(ObjectAdmin):
        form_display = ['amplitude', 'phase', 'chart']
        field_attributes = dict(amplitude = dict(delegate=delegates.FloatDelegate,
                                                 editable=True),
                                phase = dict(delegate=delegates.FloatDelegate,
                                             editable=True),
                                chart = dict(delegate=delegates.ChartDelegate) )
```

The **PlotContainer** object takes as its arguments, the same arguments that can be passed to the matplotlib plot command. The container stores all those arguments, and later passes them to the plot command executed within the gui thread.

doc/../_static/snippets/simple_plot.png

The simpel chart containers map to their respective matplotlib command. They include :

3.9.2 Actions

The *PlotContainer* and *BarContainer* can be used to print or display charts as part of an action through the use of the appropriate action steps :

- camelot.view.action_steps.print_preview.PrintChart
- camelot.view.action_steps.gui.ShowChart

```
    class ChartPrint( Action ):

        def model_run( self, model_context ):
            from camelot.container.chartcontainer import BarContainer
            from camelot.view.action_steps import PrintChart
            chart = BarContainer( [1, 2, 3, 4],
                                  [5, 1, 7, 2] )
            print_chart_step = PrintChart( chart )
```

```
        print_chart_step.page_orientation = QtGui.QPrinter.Landscape
        yield print_chart_step
```

3.9.3 Advanced Plots

For more advanced plots, the `camelot.container.chartcontainer.AxesContainer` class can be used. The *AxesContainer* class can be used as if it were a matplotlib *Axes* object. But when a method on the *AxesContainer* is called it will record the method call instead of creating a plot. These method calls will then be replayed by the gui to create the actual plot.

```
from camelot.admin.object_admin import ObjectAdmin
from camelot.view.controls import delegates
from camelot.container.chartcontainer import AxesContainer

class Wave(object):

    def __init__(self):
        self.amplitude = 1
        self.phase = 2.89

    @property
    def chart(self):
        import math
        axes = AxesContainer()
        x_data = [x/100.0 for x in range(1, 700, 1)]
        y_data = [self.amplitude * math.sin(x - self.phase) for x in x_data]
        axes.plot( x_data, y_data )
        axes.grid( True )
        axes.axvspan(self.phase-0.05, self.phase+0.05, facecolor='b', alpha=0.5)
        return axes

    class Admin(ObjectAdmin):
        form_display = ['amplitude', 'phase', 'chart']
        field_attributes = dict(amplitude = dict(delegate=delegates.FloatDelegate,
                                                 editable=True),
                                phase = dict(delegate=delegates.FloatDelegate,
                                             editable=True),
                                chart = dict(delegate=delegates.ChartDelegate) )
```

3.9.4 More

For more information on the various types of plots that can be created, have a look at the Matplotlib Gallery.

When the AxesContainer does not provide enough flexibility, for example when the plot needs to manipulated through its object structure, more customization is possible by sub-classing either the `camelot.container.chartcontainer.AxesContainer` or the `camelot.container.chartcontainer.FigureContainer`:

3.10 Document Management

Camelot provides some features for the management of documents. Notice that documents managed by Camelot are stored in a specific location (either an application directory on the local disk, a network share or a remote server).

This in contrast with some application that just store the link to a file in the database, and don't store the file itself.

Three concepts are important for understanding how Camelot handles documents :

- The **Storage** : this is the place where Camelot stores its documents, by default this is a directory on the local system. When a file is checked in into a storage, a StoredFile is returned. Files are checked out from the storage by their StoredFile representation.
- The **StoredFile** : a stored file is a representation of a file stored in a storage. It does not contain the file itself but its name and meta information.
- The **File** Field type : is a custom field type to write and read the StoredFile into the database. The actual name of the StoredFile is the only thing stored in the database.

3.10.1 The File field type

Usually the first step when working with documents is to use the File field type somewhere in the model definition. Alternatively the Image field type can be used if one only wants to store images in that field.

3.10.2 The StoredFile

When the File field type is used in the code, it returns and accepts objects of type StoredFile.

The Image field type will return objects of type StoredImage.

3.10.3 The Storage

This is where the actual file is stored. The default storage implementation simply represents a directory on the file system.

3.11 Under the hood

A lot of things happen when a Camelot application starts up. In this section we give a brief overview of those which might need to be adapted for more complex applications

3.11.1 Global settings

Camelot has a global *settings* object of which the attributes are used throughout Camelot whenever a piece of global configuration is needed. Examples of such global configuration are the location of the database and the location of stored files and images. To access the global configuration, simply import the object

```
from camelot.core.conf import settings
print settings.CAMELOT_MEDIA_ROOT()
```

To manipulate the global configuration, create a class with the needed attributes and methods and append it to the global configuration :

The *settings* object should have a method named `ENGINE`, uses the create_engine SQLAlchemy function to create a connection to the database. Camelot provides a default `sqlite` URI scheme. But you can set your own.

```
def ENGINE( self ):
    from sqlalchemy import create_engine
    return create_engine(u'sqlite:///%s/%s'%( self.data_folder,
                                              self.data ) )
```

Older versions of Camelot looked for a *settings* module on *sys.path* to look for the global configuration. This approach is still supported.

3.11.2 Setting up the ORM

When the application starts up, the *setup_model* method of the *Settings* class is called. In this function, all model files should be imported, to make sure the model has been completely setup. The importing of these files is enough to define the mapping between objects and tables.

The import of these model definitions should happen before the call to *create_all* to make sure all models are known before the tables are created.

3.11.3 Setting up the Database

Engine

The *Settings* class should contain a method named *ENGINE* that returns a connection to the database. Whenever a connection to the database is needed, this method will be called. The `camelot.core.conf.SimpleSettings` has a default *ENGINE* method that returns an SQLite database in a user directory.

Metadata

SQLAlchemy defines the `MetaData` class. A *MetaData* object contains all the information about a database schema, such as Tables, Columns, Foreign keys, etc. The `camelot.core.sql` contains the singleton *metadata* object which is the default `MetaData` object used by Camelot. In the *setup_model* function, this *metadata* object is bound to the database engine.

In case an application works with multiple database schemas in parallel, this step needs to be adapted.

Creating the tables

By simply importing the modules which contain parts of the model definition, the needed table information is added to the *metadata* object. At the end of the *setup_model* function, the *create_all* method is called on the metadata, which will create the tables in the database if they don't exist yet.

3.11.4 Working without the default model

Camelot comes with a default model for Persons, Organizations, History tracking, etc.

To turn these on or off, simply add or remove the import statements of those modules from the *setup_model* method in the *Settings* class.

3.11.5 Transactions

Transactions in Camelot can be used just as in normal SQLAlchemy. This means that inside a `camelot.admin.action.base.Action.model_run()` method a transaction can be started and committed

```
with model_context.session.begin()
    ...do some modifications...
```

More information on the transactional behavior of the session can be found in the SQLAlchemy documentation ...

3.11.6 Using Camelot without the GUI

Often a Camelot application also has a non GUI part, like batch scripts, server side scripts, etc.

It is of course perfectly possible to reuse the whole model definition in those non GUI parts. The easiest way to do so is to leave the Camelot GUI application as it is and then in the non GUI script, initialize the model first

```
from camelot.core.conf import settings
settings.setup_model()
```

From that point, all model manipulations can be done. Access to the single session can be obtained from anywhere through the *Session* factory method

```
from camelot.core.orm import Session
session = Session()
```

After the manipulations to the model have been done, they can be flushed to the db

```
session.flush()
```

3.12 Built in data models

Camelot comes with a number of built in data models. To avoid boiler plate models needed in almost any application (like Persons, Addresses, etc.), the developer is encouraged to use these data models as a start for developing custom applications.

3.12.1 Modules

The `camelot.model` module contains a number of submodules, each with a specific purpose

To activate such a submodule, the submodule should be imported in the *setup_model* method of *settings* class, before the tables are created

```
def setup_model( self ):
    from camelot.core.sql import metadata
    metadata.bind = self.ENGINE()
    from camelot.model import authentication
    from camelot.model import party
    from camelot.model import i18n
    from camelot.core.orm import setup_all
    setup_all( create_tables=True )
```

Persons and Organizations

I18N

Fixture

Authentication

Batch Jobs

A batch job object can be used as a context manager :

```
from camelot.model.batch_job import BatchJob, BatchJobType
synchronize = BatchJobType.get_or_create( u'Synchronize' )
with BatchJob.create( synchronize ) as batch_job:
    batch_job.add_strings_to_message( [ u'Synchronize part A',
                                        u'Synchronize part B' ] )
    batch_job.add_strings_to_message( [ u'Done' ], color = 'green' )
```

Whenever an exception happens inside the *with* block, the stack trace of this exception will be written to the bach job object and it's status will be set to *errors*. At the end of the *with* block, the status of the batch job will be set to *finished*.

History tracking

3.12.2 Customization

Adding fields

Sometimes the built in models don't have all the fields or relations required for a specific application. Fortunately it is possible to add fields to an existing model on a per application base.

To do so, simply assign the required fields in the application specific model definition, before the tables are created.

```
party.Person.language = schema.Column( types.Unicode(30) )

metadata.create_all()
p = party.Person( first_name = u'Peter',
                  last_name = u'Principle',
                  language = u'English' )
session.flush()
```

3.13 Fixtures : handling static data in the database

Some tables need to be filled with default data when users start to work with the application. The Camelot fixture module `camelot.model.fixture` assist in handling this kind of data.

Suppose we have an entity *PartyCategory* to divide Persons and Organizations into certain groups.

The complete definition of such an entity can be found in `camelot.model.authentication.PartyCategory`.

To make things easier for the first time user, some prefab categories should be available when the user starts the application. Such as *Suspect*, *Prospect*, *VIP*.

3.13.1 When to update fixtures

Most of the time static data should be created or updated right after the model has been set up and before the user starts using the application.

The easiest place to do this is in the setup_model method inside the settings.py module.

So we rewrite settings.py to include a call to a new update_fixtures method:

```
def update_fixtures():
  """Update static data in the database"""
  from camelot.model.fixture import Fixture
  from model import MovieType

def setup_model():
  from camelot.model import *
  from camelot.model.memento import *
  from camelot.model.synchronization import *
  from camelot.model.authentication import *
  from camelot.model.i18n import *
  from camelot.model.fixture import *
  from model import *
  setup_all(create_tables=True)
  updateLastLogin()
  update_fixtures()
```

3.13.2 Creating new data

When creating new data with the fixture module, a reference to the created data will be stored in the fixture table along with a 'fixture key'. This fixture key can be used later to retrieve or update the created data.

So lets create some new movie types:

```
def update_fixtures():
  """Update static data in the database"""
  from camelot.model.fixture import Fixture
  from model import MovieType
  Fixture.insertOrUpdateFixture(MovieType,
                                fixture_key = 'comic',
                                values = dict(name='Comic'))
  Fixture.insertOrUpdateFixture(MovieType,
                                fixture_key = 'scifi',
                                values = dict(name='Science Fiction'))
```

Fixture keys should be unique for each Entity class.

3.13.3 Update fixtures

When a new version of the application gets released, we might want to change the static data and add some icons to the movie types. Thanks to the 'fixture key', it's easy to retrieve and update the already inserted data, just modify the update_fixtures function:

```
def update_fixtures():
  """Update static data in the database"""
  from camelot.model.fixture import Fixture
  from model import MovieType
  Fixture.insertOrUpdateFixture(MovieType,
```

```
                                fixture_key = 'comic',
                                values = dict(name='Comic', icon='spiderman.png'))
Fixture.insertOrUpdateFixture(MovieType,
                                fixture_key = 'scifi',
                                values = dict(name='Science Fiction', icon='light_saber.png'))
```

3.13.4 The fixture version

In case lots of data needs to be read into the database (like a list of postal codeds), it might make no sense to create a new fixture for each code, instead a fixture version number can be set to indicate a list has been read into the database. The `camelot.model.fixture.FixtureVersion` exists to facilitate this.

```
import csv
if FixtureVersion.get_current_version( u'demo_data' ) == 0:
    reader = csv.reader( open( example_file ) )
    for line in reader:
        Person( first_name = line[0], last_name = line[1] )
    FixtureVersion.set_current_version( u'demo_data', 1 )
    session.flush()
```

3.14 Managing a Camelot project

Once a project has been created and set up as described in the tutorial *Creating a Movie Database Application*, it needs to be maintained and managed over time.

The command line tool **camelot_admin.py** exist to assist in the management of Camelot projects.

3.14.1 camelot_admin.py

3.15 The Two Threads

Most users of Camelot won't need the information in this Chapter and can simply enjoy building applications that don't freeze. However, if you start customizing your application beyond developing custom delegates, this information might be crucial to you.

3.15.1 Introduction

A very important aspect of any GUI application is the speed with which it responds to the user's request. While it is acceptable that some actions take some time to complete, an application freezing for even half a second makes the user feel uncomfortable.

From an application developer's point of view, potential freezes are everywhere (open a file, access a database, do some calculations), so we need a structural approach to get rid of them.

Two different approaches are possible. The first approach is split all possibly blocking operations into small parts and hook everything together with events. This is the approach taken in some of the QT classes (eg.: the network classes) or in the Twisted framework. The second approach is to use multiple threads of execution and make sure the blocking operations run in another thread than the GUI.

Events :

- No multi-threaded programming needed : no deadlocks etc.

- Every single library you use must support this approach

Multiple threads :

- Scary : potential race conditions and deadlocks
- Can be used with existing libraries

The Camelot framework was developed using the multi-threaded approach. This allows to build on top of a large number of existing libraries (sqlalchemy, PIL, numpy,...) that don't support the event based approach.

3.15.2 Two Threads

To keep the problems associated with multi-threaded programming under control, Camelot runs only two threads for its basic operations. Those threads don't share any data with each other and exchange information using a message queue (the way Erlang advocates). This ensures there are no deadlocks or race conditions.

The first thread, called the GUI Thread contains the QT widgets and runs the QT event loop. No blocking operations should take place in this thread. The second thread contains all the data, like objects mapped to the database by sqlalchemy, and is called the Model Thread.

This approach keeps the problem of application freezes under control, it won't speed up your application when certain actions take a long time, but it will ensure the gui remains responsive during those actions.

3.15.3 The Model Thread

Since every single operation on a data model is potentially blocking (eg : getting an attribute of a class mapped to the database by sqlalchemy might trigger a query to the database which might be overloaded at that time), the whole data model lives in a separate thread and every operation on the data model should take place within this thread.

To keep things simple and avoid the use of locks and data synchronization between threads, there is only one such thread, called the Model Thread.

Other threads that want to interact with the model can post operations to the model thread using its queue

```
from camelot.view.model_thread import get_model_thread

mt = get_model_thread()
mt.post(my_operation)
```

where 'my_operation' is a function that will then be executed within the model thread.

3.15.4 The GUI Thread

Now that all potentially blocking operations have been move to the model thread, we have a GUI Thread that never blocks. But the GUI thread will need some data from the model to present to the user.

The GUI thread gets its data by posting an operation to the Model Thread that strips some data from the model, this data will then be posted by the Model thread to the GUI thread.

Suppose we want to display the name of the first person in the database in a QLabel

```
from camelot.view.model_thread import get_model_thread
from PyQt4 import QtGui

class PersonLabel(QtGui.QLabel):

  def __init__(self):
```

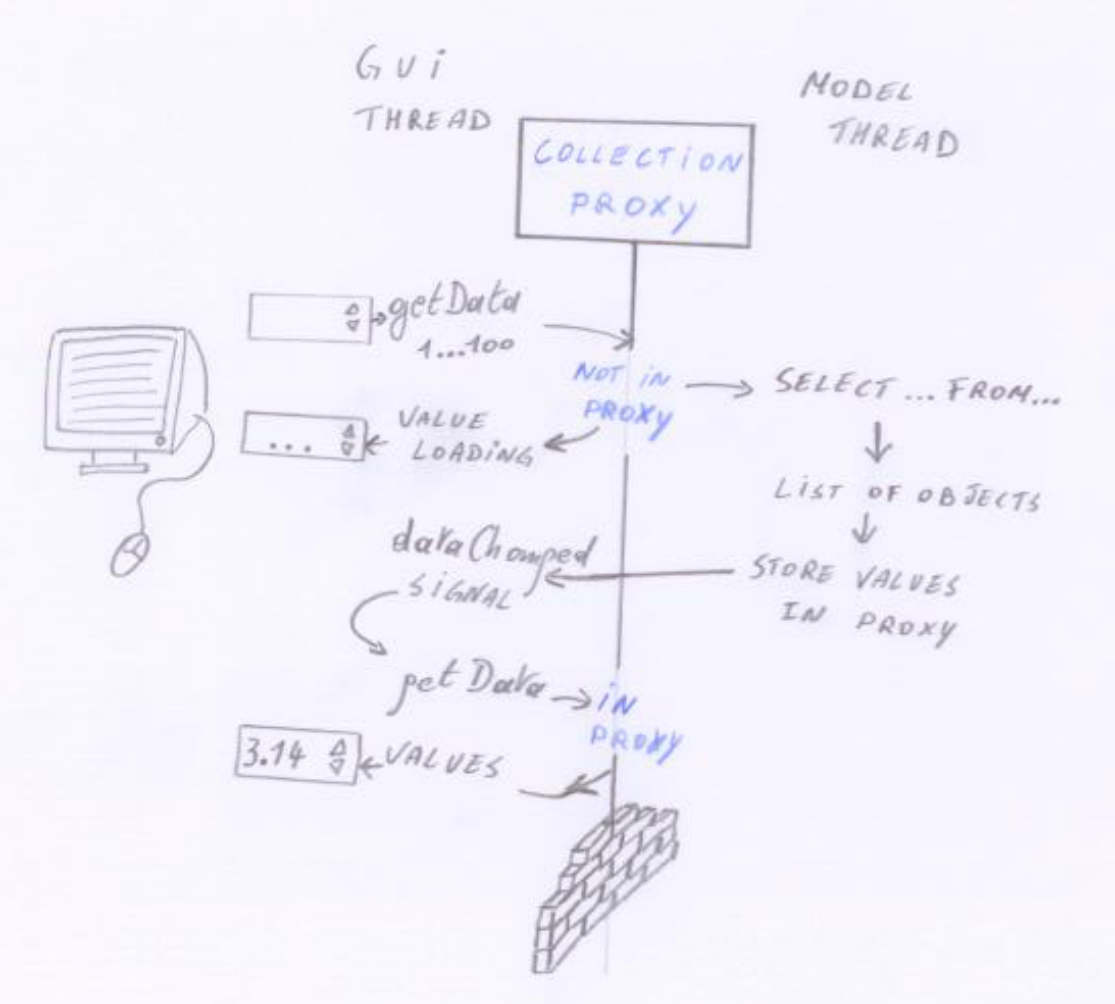
```
QtGui.QLabel.__init__(self)
mt = get_model_thread()
mt.post(self.strip_data_from_model, self.put_data_on_label)

  def strip_data_from_model(self):
from camelot.model.authentication import Person
return Person.query.first().name

  def put_data_on_label(self, name):
    self.setText(name)
```

When the strip_data_from_model method is posted to the Model Thread, it will be executed within the Model Thread and its result (the name of the person) will be posted back to the GUI thread. Upon arrival of the name in the GUI thread the function put_data_on_label will be executed within the GUI thread with as its first argument the name.

In reality, the stripping of data from the model and presenting this data to the gui is taken care off by the proxy classes in camelot.view.proxy.

3.15.5 Actions

3.15.6 Proxy classes



3.15.7 Application speedup

3.16 Frequently Asked Questions

3.16.1 How to use the PySide bindings instead of PyQt ?

The Camelot sources as well as the example videostore application can be converted from PyQt applications to PySide with the *camelot_admin* tool.

Download the sources and position the shell in the main directory, and then issue these commands:

```
python camelot/bin/camelot_admin.py to_pyside .
```

This will create a subdirectory 'to_pyside' which contains the converted source code.

3.16.2 Can I use Camelot with an existing database ?

Both Declarative and Camelot can be used with an existing schema. However, since Camelot acts on objects, the classes for those objects still need to be defined.

Here's a short example of using camelot with an existing database :

```
from sqlalchemy.engine import create_engine
from sqlalchemy.pool import StaticPool

engine = create_engine( 'sqlite:///test.sqlite' )
#
# Create a table in the database using plain old sql
#
connection = engine.connect()
try:
    connection.execute("""drop table person""")
except:
    pass
connection.execute( """create table person ( pk INTEGER PRIMARY KEY,
                                             first_name TEXT NOT NULL,
                                             last_name TEXT NOT NULL )""" )
connection.execute( """insert into person (first_name, last_name)
                       values ("Peter", "Principle")""" )

#
# Use declarative to reflect the table and create classes
#
from camelot.admin.entity_admin import EntityAdmin
from camelot.core.sql import metadata
from sqlalchemy.schema import Table
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base( metadata = metadata )

class Person( Base ):
    __table__ = Table( 'person', Base.metadata,
                       autoload=True, autoload_with=engine )
```

```
    class Admin( EntityAdmin ):
        list_display = ['first_name', 'last_name']

#
# Setup a camelot application
#
from camelot.admin.application_admin import ApplicationAdmin
from camelot.admin.section import Section
from camelot.core.conf import settings

class AppAdmin( ApplicationAdmin ):

    def get_sections( self ):
        return [ Section( 'All tables', self, items = [Person] ) ]

class Settings(object):

    def ENGINE( self ):
        return engine

    def setup_model( self ):
        metadata.bind = engine

settings.append( Settings() )
app_admin = AppAdmin()

#
# Start the application
#
if __name__ == '__main__':
    from camelot.view.main import main
    main( app_admin )
```

More information on using Declarative with an existing database schema can be found in the Declarative documentation.

3.16.3 Why is there no *Save* button ?

Early on in the development process, the controversial decision was made not to have a *Save* button in Camelot. Why was that ?

- User friendlyness. One of the major objectives of Camelot is to be user friendly. This also means we should reduce the number of 'clicks' a user has to do before achieving something. We believe the 'Save' click is an unneeded click. The application knows when the state of a form is valid for persisting it to the database, and can do so without user involvement. We also want to take the 'saving' issue out of the mind of the user, he should not bother wether his work is 'saved', it simply is.
- Technical. Once you decide to use a *Save* button, you need to ask yourself where you will put that button and what its effect will be. This question becomes difficult when you want to enable the user to edit a complex datastructure with one-to-many and many-to-many relations. Most applications solve this by limiting the options for the user. For example, most accounting packages will not allow you to create a new customer when you are creating a new invoice. Because when you save the invoice, should the customer be saved as well ? Or should the customer have it's own save button ? Those packages therefor require the user to first create a customer, and only then can an invoice be created. These are limitation we don't want to impose with Camelot.
- Consistency between editing in table or form view. We wanted the table view to be really easy to edit (to behave a bit like a spreadsheet), so it's easy for the user to do bulk updates. As such the user should not be bothered by

pressing the *Save* button all the time. If there is no need to save in the table view, there should be no need in the form view either.

Some couter arguments for this decision are :

- But what if the user wants to 'modify' a form and not save those changes ? This is indeed something that is not possible without a *Save* and it accompanying *Cancel* button. But this is something a developer will do a lot while testing an application, but is outside of the normal workflow of a user. Most users typically want to enter or modify as much data as possible, they are not testing the application to see how it would behave on certain data input.
- A form should be validated before it is saved. In an application there are two levels of validation. The first level is to validate before something is persisted into the database, this can be done in Camelot using a custom implementation of a `camelot.admin.validator.entity_validator.EntityValidator`. The second level is a validation before the entered data can be used in the business process. To do this second level validation, one can use state changes (Action buttons that change the state of a form, eg from 'Draft' to 'Complete'). A good example of this is when entering a booking into an accounting package. When a booking is entered, it can only be used when debit equals credit. What would happen when this validation is done at the moment the form is 'saved'. Suppose a user has been working for the better part of the day on a complex booking, but is not done yet at the end of the day. Since he cannot yet save his work he has two options, discard it and restart the next day, or enter some bogus data to be able to save it. What will happen in the later case when his manager is creating a report a bit later. So the correct situation in this case is having your work saved at all times, and to put your booking from a 'draft' state to a 'complete' state once its ready. This state change will then check if debit equals credit.

Two years after we made this move, Apple decided to follow our example : http://www.apple.com/macosx/whats-new/auto-save.html

3.16.4 But my users really want a *Save* button ?

We advise you to listen very well to the arguments the user has for wanting a *Save* button. You will be able to solve most of them by using state changes instead of a *Save* button. The other arguments probably have to do with expections users have from using other applications, as for those simply ask the users to try to work for a week without a *Save* button and get back to you if after that week, they still have issues with it. Please let us know when they do !

CHAPTER

FOUR

MIGRATE EXISTING CAMELOT PROJECTS

4.1 Migrate from Camelot 11.12.30 to 12.06.29

The place of the default *metadata* has changed. So the top line at the model files should change from:

```
from camelot.model import metadata
```

to:

```
from camelot.core.sql import metadata
```

All Camelot models that you wish to use should be explicitely imported in the *setup_model* method in *settings.py*. And the metadata should be bound to the engine explicitly in the *setup_model* method:

```
def setup_model():
    from camelot.core.sql import metadata
    metadata.bind = ENGINE()
    from camelot.model import authentication
    from camelot.model import party
    from camelot.model import i18n
    from camelot.model import memento
    from camelot.model import fixture
    setup_model( True )
```

The *authentication* module has been split into *authentication* and *party*. *Person* and *Organization* related imports should be redefined

```
from camelot.model.authentication import Person
```

Should become

```
from camelot.model.party import Person
```

There were some changes in the data model of Camelot, in the parts that track change history and handle authentication. Run this SQL script against your database to do the upgrade, after taking a backup.

On Postgresql

```
ALTER TABLE memento ADD memento_type INT;
ALTER TABLE memento ADD COLUMN previous_attributes bytea;
UPDATE memento SET
    memento_type = 1,
```

```
    previous_attributes = memento_update.previous_attributes
FROM memento_update WHERE memento.id = memento_update.memento_id;
UPDATE memento SET
    memento_type = 2,
    previous_attributes = memento_delete.previous_attributes
FROM memento_delete WHERE memento.id = memento_delete.memento_id;
UPDATE memento SET
    memento_type = 3
FROM memento_create WHERE memento.id = memento_create.memento_id;
ALTER TABLE memento ALTER COLUMN memento_type SET NOT NULL;
ALTER TABLE memento DROP COLUMN row_type;
DROP TABLE memento_update;
DROP TABLE memento_delete;
DROP TABLE memento_create;
CREATE INDEX ix_memento_memento_type
    ON memento (memento_type);
ALTER TABLE authentication_mechanism ADD COLUMN authentication_type INT;
ALTER TABLE authentication_mechanism ADD COLUMN username VARCHAR(40);
ALTER TABLE authentication_mechanism ADD COLUMN password VARCHAR(200);
ALTER TABLE authentication_mechanism ADD COLUMN from_date DATE;
ALTER TABLE authentication_mechanism ADD COLUMN thru_date DATE;
ALTER TABLE authentication_mechanism DROP COLUMN row_type;
ALTER TABLE authentication_mechanism DROP COLUMN is_active;
UPDATE authentication_mechanism SET
    authentication_type = 1,
    from_date = '2000-01-01',
    thru_date = '2400-12-31',
    username = authentication_mechanism_username.username,
    password = authentication_mechanism_username.password
FROM authentication_mechanism_username WHERE authentication_mechanism.id = authentication_mechanism_u
ALTER TABLE authentication_mechanism ALTER COLUMN authentication_type SET NOT NULL;
ALTER TABLE authentication_mechanism ALTER COLUMN from_date SET NOT NULL;
ALTER TABLE authentication_mechanism ALTER COLUMN thru_date SET NOT NULL;
DROP TABLE authentication_mechanism_username;
CREATE INDEX ix_authentication_mechanism_from_date
    ON authentication_mechanism (from_date);
CREATE INDEX ix_authentication_mechanism_thru_date
    ON authentication_mechanism (thru_date);
CREATE INDEX ix_authentication_mechanism_username
    ON authentication_mechanism (username);
CREATE INDEX ix_authentication_mechanism_authentication_type
    ON authentication_mechanism (authentication_type);
```

On MySQL

```
ALTER TABLE memento ADD memento_type INT;
ALTER TABLE memento ADD COLUMN previous_attributes blob;
UPDATE memento, memento_update SET
    memento.memento_type = 1,
    memento.previous_attributes = memento_update.previous_attributes
WHERE memento.id = memento_update.memento_id;
UPDATE memento, memento_delete SET
    memento.memento_type = 2,
    memento.previous_attributes = memento_delete.previous_attributes
WHERE memento.id = memento_delete.memento_id;
UPDATE memento, memento_create SET
    memento.memento_type = 3
WHERE memento.id = memento_create.memento_id;
ALTER TABLE memento ALTER COLUMN memento_type SET NOT NULL;
```

```
ALTER TABLE memento DROP COLUMN row_type;
DROP TABLE memento_update;
DROP TABLE memento_delete;
DROP TABLE memento_create;
CREATE INDEX ix_memento_memento_type
    ON memento (memento_type);
ALTER TABLE authentication_mechanism ADD COLUMN authentication_type INT;
ALTER TABLE authentication_mechanism ADD COLUMN username VARCHAR(40);
ALTER TABLE authentication_mechanism ADD COLUMN password VARCHAR(200);
ALTER TABLE authentication_mechanism ADD COLUMN from_date DATE;
ALTER TABLE authentication_mechanism ADD COLUMN thru_date DATE;
ALTER TABLE authentication_mechanism DROP COLUMN row_type;
ALTER TABLE authentication_mechanism DROP COLUMN is_active;
UPDATE authentication_mechanism, authentication_mechanism_username SET
    authentication_mechanism.authentication_type = 1,
    authentication_mechanism.from_date = '2000-01-01',
    authentication_mechanism.thru_date = '2400-12-31',
    authentication_mechanism.username = authentication_mechanism_username.username,
    authentication_mechanism.password = authentication_mechanism_username.password
WHERE authentication_mechanism.id = authentication_mechanism_username.authenticationmechanism_id;
ALTER TABLE authentication_mechanism ALTER COLUMN authentication_type SET NOT NULL;
ALTER TABLE authentication_mechanism ALTER COLUMN from_date SET NOT NULL;
ALTER TABLE authentication_mechanism ALTER COLUMN thru_date SET NOT NULL;
DROP TABLE authentication_mechanism_username;
CREATE INDEX ix_authentication_mechanism_from_date
    ON authentication_mechanism (from_date);
CREATE INDEX ix_authentication_mechanism_thru_date
    ON authentication_mechanism (thru_date);
CREATE INDEX ix_authentication_mechanism_username
    ON authentication_mechanism (username);
CREATE INDEX ix_authentication_mechanism_authentication_type
    ON authentication_mechanism (authentication_type);
```

Or simply drop these tables and have them recreated by Camelot and lose the history information

```
DROP TABLE memento_update;
DROP TABLE memento_delete;
DROP TABLE memento_create;
DROP TABLE memento;
DROP TABLE authentication_mechanism_username;
DROP TABLE authentication_mechanism;
```

Consider converting your *settings.py* module to a *settings object* .

4.2 Migrate from Camelot 12.06.29 to 13.04.13

- Replace all imports from *elixir* with import from *camelot.core.orm*. This should cover most use cases of Elixir, use cases that are not covered in the new module (inheritance, elixir extensions) should be rebuild using Declarative. Notice that it is still possible to continue using Elixir, but not encouraged. This is a good time to move your code base over to Declarative.
- If the *embedded=True* field attribute is in use, this should be removed, as it is no longer supported. The proposed alternative is to use the `camelot.admin.object_admin.ObjectAdmin.get_compounding_objects()` method on the admin to display multiple objects in the same form.
- Database migration commands for the changed batch job model:

```
CREATE TABLE `batchjob_status` (
  `status_datetime` date DEFAULT NULL,
  `status_from_date` date DEFAULT NULL,
  `status_thru_date` date DEFAULT NULL,
  `from_date` date NOT NULL,
  `thru_date` date NOT NULL,
  `classified_by` int(11) NOT NULL,
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `status_for_id` int(11) NOT NULL,
  PRIMARY KEY (`id`),
  KEY `status_for_id` (`status_for_id`),
  KEY `ix_batchjob_status_classified_by` (`classified_by`),
  CONSTRAINT `batchjob_status_ibfk_1` FOREIGN KEY (`status_for_id`) REFERENCES `batch_job` (
);
ALTER TABLE `batch_job` DROP COLUMN `status`;
```

- Database migration commands for the changed authentication model:

```
CREATE TABLE authentication_group
(
  name character varying(256) NOT NULL,
  id serial NOT NULL,
  CONSTRAINT authentication_group_pkey PRIMARY KEY (id )
)

CREATE TABLE authentication_group_member
(
  authentication_group_id integer NOT NULL,
  authentication_mechanism_id integer NOT NULL,
  CONSTRAINT authentication_group_member_pkey PRIMARY KEY (authentication_group_id , authent
  CONSTRAINT authentication_group_members_fk FOREIGN KEY (authentication_group_id)
      REFERENCES authentication_group (id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION,
  CONSTRAINT authentication_group_members_inverse_fk FOREIGN KEY (authentication_mechanism_i
      REFERENCES authentication_mechanism (id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION
)

CREATE TABLE authentication_group_role
(
  role_id serial NOT NULL,
  group_id integer NOT NULL,
  CONSTRAINT authentication_group_role_pkey PRIMARY KEY (role_id , group_id ),
  CONSTRAINT authentication_group_role_group_id_fkey FOREIGN KEY (group_id)
      REFERENCES authentication_group (id) MATCH SIMPLE
      ON UPDATE CASCADE ON DELETE CASCADE
)
```

CHAPTER

FIVE

ADVANCED TOPICS

This is documentation for advanced usage of the Camelot library.

5.1 Internationalization

The Camelot translation system is a very small wrapper around the Qt translation system. Internally, it uses the `QCoreApplication.translate()` method to do the actual translation.

On top of that, it adds the possibility for end users to change translations theirselves. Those translations are stored in the database. This mechanism can be used to adapt the vocabulary of an application to that of a specific company.

5.1.1 How to Specify Translation Strings

Translation strings specify "This text should be translated.". It's your responsibility to mark translatable strings; the system can only translate strings it knows about.

```
from camelot.core.utils import ugettext as _

message = _("Hello brave new world")
```

The above example translates the given string immediately. This is not always desired, since the message catalog might not yet be loaded at the time of execution. Therefore translation strings can be specified as lazy. They will only get translated when they are used in the GUI.

```
from camelot.core.utils import ugettext_lazy as _

message = _("This translation is delayed")
```

Translation strings in model definitions should always be specified as lazy translation strings. Only lazy translation strings can be translated by the end user in various forms.

5.1.2 Translating Camelot itself

To extract translation files from the Camelot source code, Babel needs to be installed.

In the root folder of the Camelot source directory.

First update the translation template:

```
python setup.py extract_messages
```

If your language directory does not yet exists in 'camelot/art/translations':

```
python setup.py init_catalog --locale nl
```

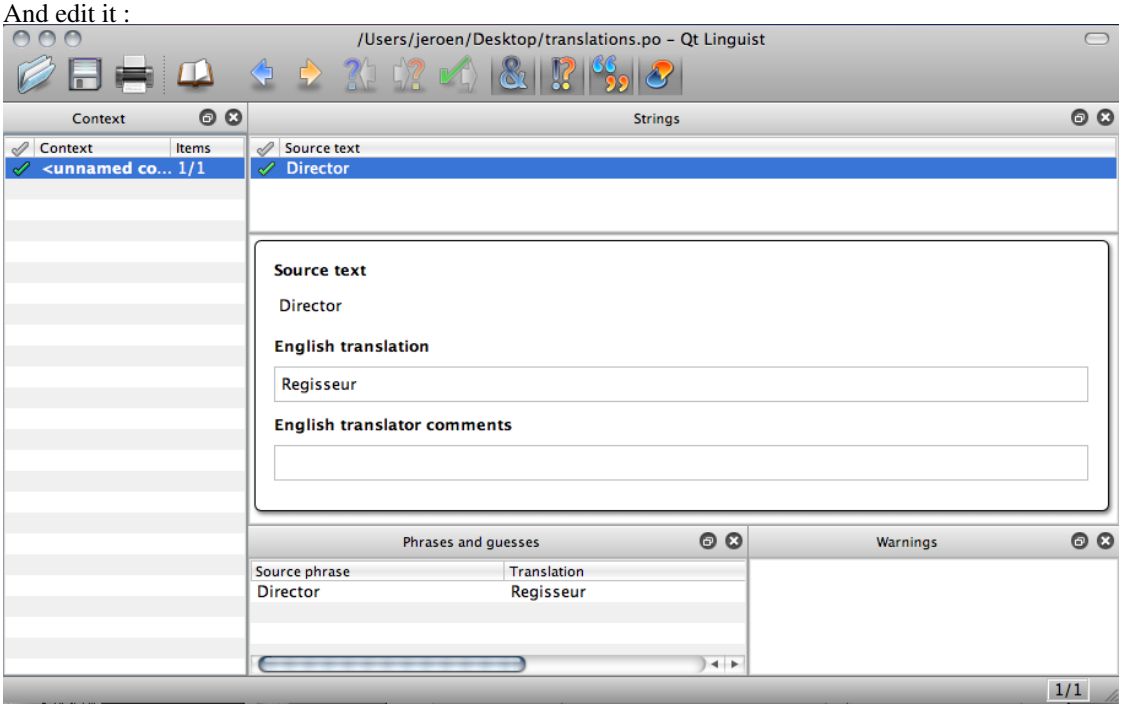
If it allready exists, update it from the translation template:

```
python setup.py update_catalog
```

In the language subdirectory of 'camelot/art/translations', there is a subdirectory 'LC_MESSAGES' which contains the .po translation file. This translation file can then be translated with linguist

```
linguist camelot.po
```

And edit it :



Make sure to save them back as GNU gettext .po files.

Then the .po file should be converted to a .qm file to make it loadable at run time:

```
lrelease camelot.po
```

Don't forget to post your new .po file on the mailing list, so it can be included in the next release.

For more background information, please have a look at the Babel Documentation

5.1.3 Where to put Translations

Translations can be put in 2 places :

- in po files which have to be loaded at application startup
- in the Translation table : this table is editable by the users via the Configuration menu. This is the place to put translations that should be editable by the users. At application startup, all records in this table related to the current language will be put in memory.

5.1.4 Loading translations

Translations are loaded when the application starts. To enforce the loading of the correct translation file, one should overwrite the `camelot.admin.application_admin.ApplicationAdmin.get_translator()` method. This method should return the proper `QtCore.QTranslator` object.

5.1.5 End user translations

Often it is convenient to let the end user create or update the translations of an application, this allows the end user to put a lot of domain knowledge into the application.

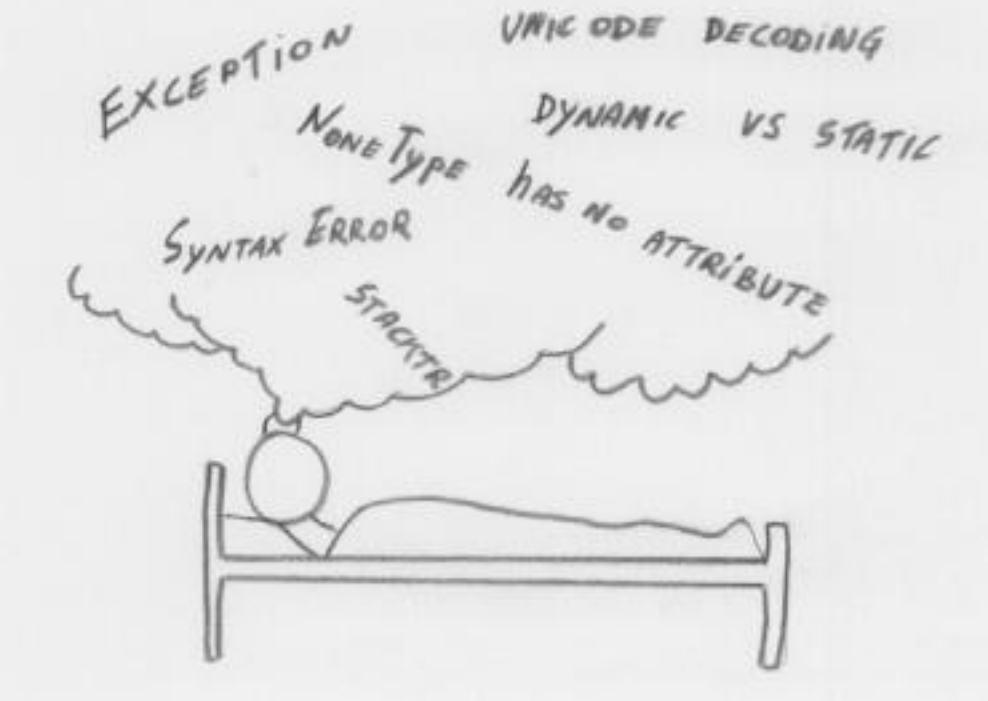
Therefore, all lazy translation strings can be translated by the end user. When the user right-clicks on a label in a form, he can select *Change translation* from the menu and update the current translation (for the current language). This effectively updates the content of the **Translation** table.

After some time the developer can take a copy of this table and decide to put these translations in po files.

5.2 Unittests

Release default

Date April 23, 2013



5.3 Deployment

After developing a Camelot application comes the need to deploy the application, either at a central location or in a distributed setup.

5.3.1 Building .egg files

Whatever the deployment setup is, it is almost always a good idea to distribute your application as a single .egg file, containing as much as possible the dependencies that are likely to change often during the lifetime of the application. Resource files (like icons or templates can be included in this .egg file as well).

Building .egg files is a relatively straightforward process using setuptools.

When a new Camelot project was created with *camelot_admin*, a `setup.py` file was made that is able to build eggs using this command

```
python -O setup.py bdist_egg --exclude-source-files
```

Note: The advantage of using .egg files comes when updating the application, simply replacing a single .egg file at a central location is enough to migrate all your users to the new version.

5.3.2 Windows deployment

Through CloudLaunch

CloudLaunch is a service to ease the deployment and update process of Python applications. It's main features are :

- Building Windows Installers
- Updating deployed applications
- Monitoring of deployed applications

As CloudLaunch is build on top of setuptools, it works with .egg files, CloudLaunch works cross platform, so it's perfectly possible to build a Windows installer, or update a Windows application from Linux.

To build a .egg file that can be deployed through CloudLaunch, use the command:

```
python.exe setup.py bdist_cloud
```

This will create 2 files in the dist/cloud folder, a traditional .egg file and a .cld file. The .egg file is a normal .egg file with some additional metadata included, and without sources. The .cld file contains metadata of the .egg file, such as its checksum, and information on how get updated versions of the .egg once deployed.

To make sure the application will run smoothly once deployed, one should test if the generated .egg and .cld combination works:

```
cd dist\cloud
cloudlaunch.exe --cld-file movie_store.cld
cd ..\..
```

If this is working, a Windows installer can be build:

```
python.exe setup.py bdist_cloud wininst_cloud
```

This will generate a movie_store.exe file in distcloud, which is an installer for your application. The end user can now install and run your application on his machine.

Now is the time to monitor the application as it runs on the end user machine:

```
python.exe setup.py monitor_cloud
```

Will display all the logs issued on the end user machine if that machine is connected to the internet.

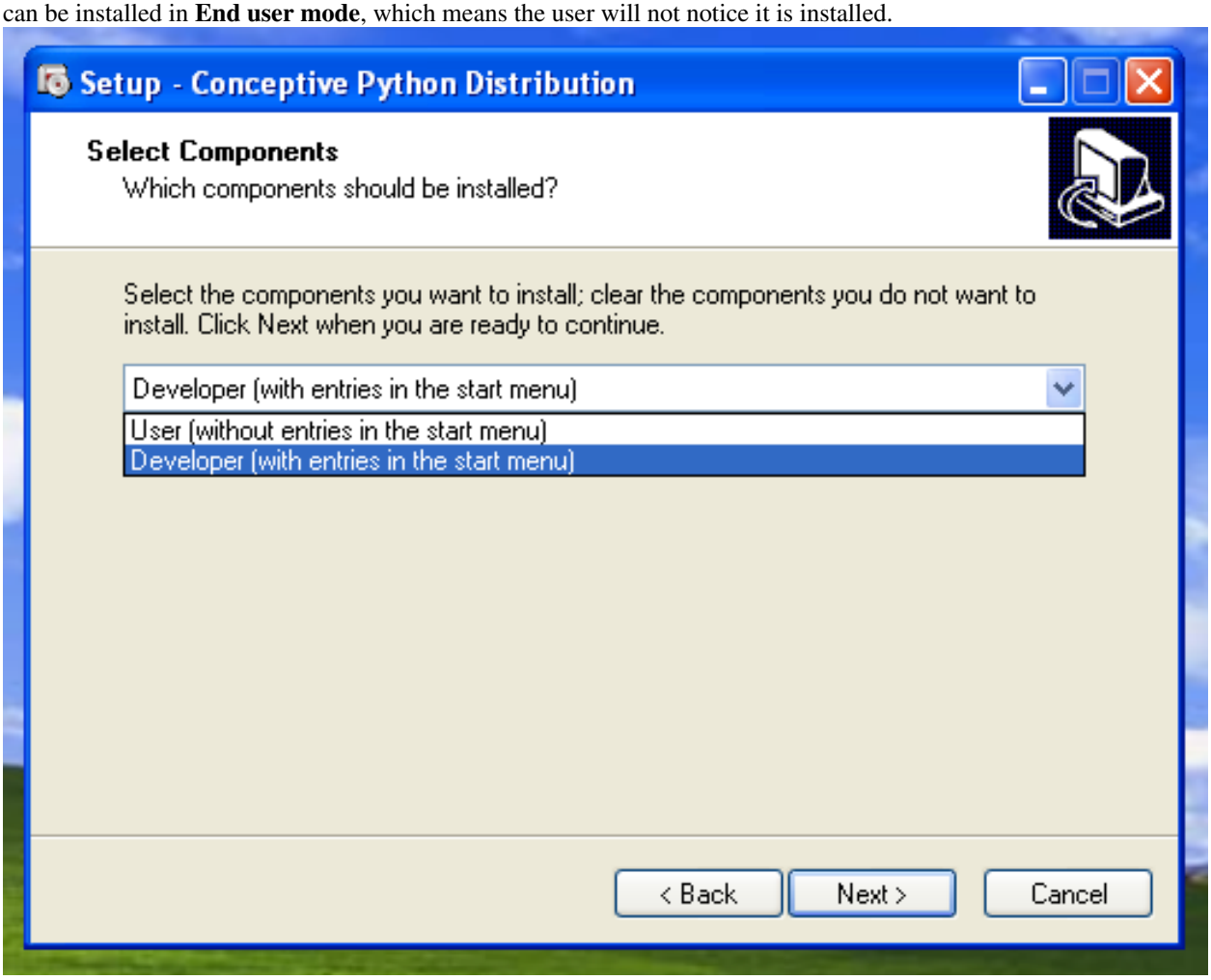
When development of the application continues, it will be needed to present the user with an updated version of the application. This is done with the command:

```
python.exe setup.py bdist_cloud upload_cloud
```

This will send an updated .egg and .cld file to the central repository, where the end-user application will check for updates. If such an update is detected, the application will download the new egg and run from that one.

Using .egg files

First of all python needs to be available on the machines that are going to run the application. The easies way to achieve this is by installing the Conceptive Python Distribution (CPD) on the target machine. This Python distribution can be installed in **End user mode**, which means the user will not notice it is installed.



Notice that for python to be available, it not necessarily needs to be installed on every machine that runs the application. Installing python on a shared disk of a central server might just be enough.

Also put the .egg file on a shared drive.

Then, the easiest way to proceed is to put a little .vbs bootstrap script on the shared drive and put shortcuts to it on the desktops of the users. The .vbs script can look like this:

```
Set WshShell = WScript.CreateObject("WScript.Shell")
WshShell.Environment("Process").item("PYTHONPATH") = "R:\movie_store-01.01-py2.7.egg;"
WshShell.Run """C:\Program Files\CPD\pythonw.exe"" -m movie_store.main"
```

5.3.3 Linux deployment

The application can be launched by putting the .egg in the PYTHONPATH and starting python with the -m option:

```
export PYTHONPATH = /mnt/r/movie_store-01.01-py2.7.egg
python.exe -m movie_store.main
```

Don't forget that all dependencies for your application should be installed on the system or put in the PYTHONPATH

5.4 Authentication and permissions

fine grained authentication and authorization is not yet included as part of the Camelot framework.

what is included is the function :

`camelot.model.authentication.get_current_authentication()`

which returns an object of type :class:'camelot.model.authentication.AuthenticationMechanism

where the username is the username of the currently logged in user (because on most desktop apps, you don't want a separate login process for your app, but rely on that of the OS).

this function can then be used if you build the *Admin* classes for your application :

- set the *editable* field attribute to a function that only returns Thrue when the current authentication requires editing of fields
- in the *ApplicationAdmin.get_sections method*, to hide/show sections depending on the logged in user
- in the *EntityAdmin* subclasses, in the *get_field_attributes* method, to set fields to editable=False/True depending on the logged in user

5.5 Development Guidlines

Date April 23, 2013

5.5.1 Python, PyQt and Qt objects

Python and Qt both have their own way of tracking objects and deleting them when they are no longer needed :

- Python does reference counting supported by a garbage collector.
- Qt has parent child relations between objects. When a parent object is deleted, all its child objects are deleted as well.

PyQt merges these two concepts by introducing **ownership** of objects :

- Pure python objects are owned by Python, Python takes care of their deletion.
- Qt objects wrapped by Python are either:
  - owned by Qt when they have a parent object, Qt will delete them, when their parent object is deleted

  - owned by Python when they have no parent, Python will delete them, and trigger the deletion of all their children by Qt

- Qt objects that are not wrapped by Python, those are in one way or another children of a Qt object that is wrapped by Python, they will get deleted by Qt.

The difficult case in this scheme is the case where Qt objects are wrapped by Python but have a parent object. This can happen in two ways :

- A Qt object is created in python, but with a parent

```
from PyQt4 import QtCore

parent = QtCore.QObject()
child = QtCore.QObject(parent=parent)
```

  In this case PyQt is able to track when the object is deleted by Qt and raises exceptions accordingly when a method of underlying Qt object is called after the deletion

```
parent = QtCore.QObject()
child = QtCore.QObject(parent=parent)
del parent
print child.objectName()
```

  will raise a RuntimeError: underlying C/C++ object has been deleted.

- A Qt object is returned from a Qt function that created the object without Python being aware of it. When the object is passed as a return value PyQt will wrap it as a Python object, but is unable to track when Qt deletes it

```
from PyQt4 import QtGui
app = QtGui.QApplication([])
window = QtGui.QMainWindow()
statusbar = window.statusBar()
del window
statusbar.objectName()
```

  Will result in a segmentation fault.

A segmentation fault will happen in several cases :

- Python tries to delete a Qt object already deleted by Qt
- PyQt calls a function of a Qt object already deleted
- Qt calls a function of a Qt object already deleted by Python

In principle, PyQt is able to handle all cases where the object has been created by Python. However, when this ownership tracking is combined with threading and signal slot connections, a lot of corner cases arise in both Qt and PyQt.

To play on safe, these guidelines are used when developing Camelot :

- Never keep a reference to objects created by Qt having a parent, so only use:

```
window.statusBar().objectName()
```

- Keep references to Qt child objects as short as possible, and never beyond the scope of a method call. This is possible because qt allows objects to have a name.

  so instead of doing

```
from PyQt4 import QtGui

class Parent( QtGui.QWidget ):
```

```
        def __init__( self ):
                super(Parent, self).__init__()
                self._child = QtGui.QLabel( parent=self )

        def do_something( self ):
                print self._child.objectName()
```

this is done

```
from PyQt4 import QtGui

class Parent( QtGui.QWidget ):

        def __init__( self ):
                super(Parent, self).__init__()
                child = QtGui.QLabel( parent=self )
                child.setObjectName( 'label' )

        def do_something( self ):
            child = self.findChild( QtGui.QWidget, 'label' )
            if child != None:
                        print child.objectName()
```

should the child object have been deleted by Qt, the findChild method will return None, and a segmentation fault is prevented. An explicit check for None is needed, since even if the widget exists, it might evaluate to 0 or an empty string.

5.6 Debugging Camelot and PyQt

5.6.1 Log the SQL Queries

Configure SQLAlchemy to log all queries:

```
logging.getLogger('sqlalchemy.engine').setLevel(logging.DEBUG)
```

5.6.2 Enable core dumps

Linux

For older gdb versions (pre 7), copy the gdbinit file from the python Misc folder:

```
cp gdbinit ~/.gdbinit
```

use:

```
ulimit -c unlimited
```

load core file in gdb:

```
gdb /usr/bin/python -c core
```

In newer gdb versions, Python can run inside gdb:

http://bugs.python.org/issue8032

To give gdb python super powers:

```
(gdb) python
>import sys
>sys.path.append('Python-2.7.1/Tools/gdb/libpython.py')
>import libpython
>reload(libpython)
>
>end
```

https://fedoraproject.org/wiki/Features/EasierPythonDebugging

Windows

- Install *Debugging tools for Windows* from MSDN

Install 'Debug Diagnostic Tool'

http://stackoverflow.com/questions/27742/finding-the-crash-dump-files-for-a-c-app

http://blogs.msdn.com/b/tess/

Setup Qt Creator

http://doc.qt.nokia.com/qtcreator-snapshot/creator-debugger-engines.html

- Install Windows Sysinternals process utilities from MSDN

http://technet.microsoft.com/en-us/sysinternals/bb795533

CHAPTER

SIX

CAMELOT ENHANCEMENT PROPOSALS

This section contains proposals to enhance Camelot. The functionallity described here might not yet be implemented. The purpose of these documents is to discuss upcomming functions and new API's before they are implemented.

6.1 Unified Model Definition

status : draft

Note: This Camelot enhancement proposal is a work in progress and implementation has not started.

6.1.1 Introduction

When Camelot is used to display objects that are mapped to the database through SQLAlchemy, Camelot uses introspection to create default views.

When displaying objects that are not mapped to the database, such introspection is not possible. This often leads to a rather verbose definition of the model and the view

```
class Task( object ):

    def __init__( self ):
        self.description = ''
        self.creation_date = datetime.date.today()

    class Admin( ObjectAdmin ):
        list_display = ['description', 'due_date']
        field_attributes = { 'description': {'delegate':delegates.TextLineDelegate,
                                             'editable':True},
                             'due_date': {'delegate':delegates.DateDelegate,
                                          'editable':True}, }
```

This proposal aims to find a way to create a less descriptive way to define model and view in the case of simple Python objects.

6.1.2 Summary

Fields on objects can be defined in a uniform way wether they are mapped to the database or not. The definition of the unmapped *Task* class would be

```
class Task( object ):
    description = Field( unicode, default = 0 )
    due_date = Field( datetime.date, default = 0 )
```

While the definition of the mapped *Task* class would be

```
class Task( Entity ):
    description = Field( sqlalchemy.types.Unicode, default = 0 )
    due_date = Field( sqlalchemy.types.Date, default = 0 )
```

Both definitions should be enough for Camelot to create a view and make the object usable in the model.

6.1.3 Fields

6.1.4 Default views

6.1.5 Field attributes

6.1.6 Relations

CHAPTER

SEVEN

SUPPORT

7.1 Community

Community support is available on the mailing list. Camelot is on Bitbucket to lower contribution efforts.

7.2 Commercial

Commercial support and training is available from Conceptive Engineering, the main authors of Camelot :

Conceptive Engineering
L Van Bauwelstraat 16
2222 Heist-op-den-Berg
Belgium

info@conceptive.be
http://www.conceptive.be
VAT BE 0878 169 209

Priority support tickets can be purchased from the shop. Please contact us for support contracts.

Indices and tables:

- *genindex*
- *modindex*
- *search*

Others:

7.2.1 Camelot Documentation contents

Tutorials

This section contains various tutorials that will help the reader get a feeling of Camelot. We assume that the reader has some knowledge of Python.

The reader can read the following sub-sections in any order.

Creating a Movie Database Application

In this tutorial we will create a fully functional movie database application with Camelot. We assume Camelot is properly *installed*. An all in one installer for Windows is available as an SDK to develop Camelot applications (Python SDK).

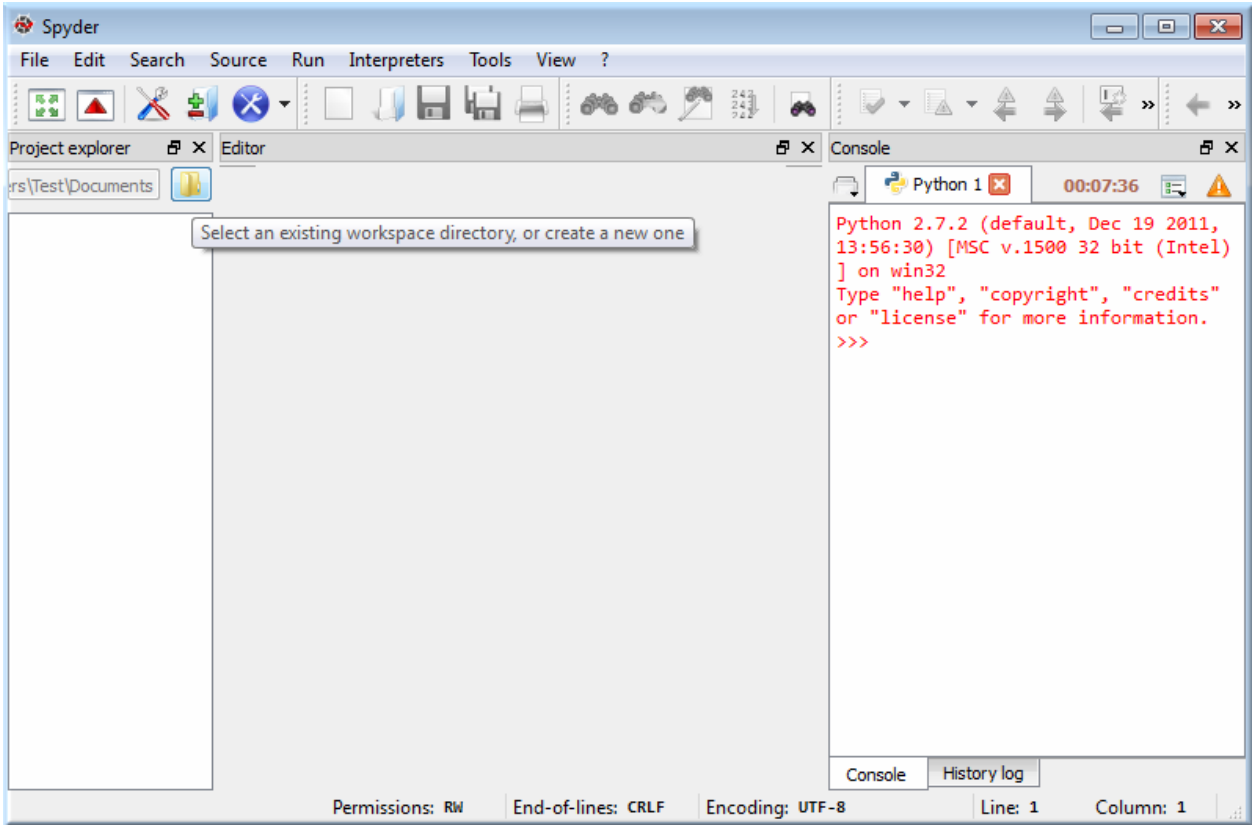
Setup Spyder In this section, we will explain how to setup the **Spyder IDE** for developing a **Camelot** project. If you are not using **Spyder**, you can skip this and jump to the next *section*.

Start → All Programs → Python SDK → Spyder

Within **Spyder**, open the *Project Explorer* :

View → Windows and toolbars → Project explorer

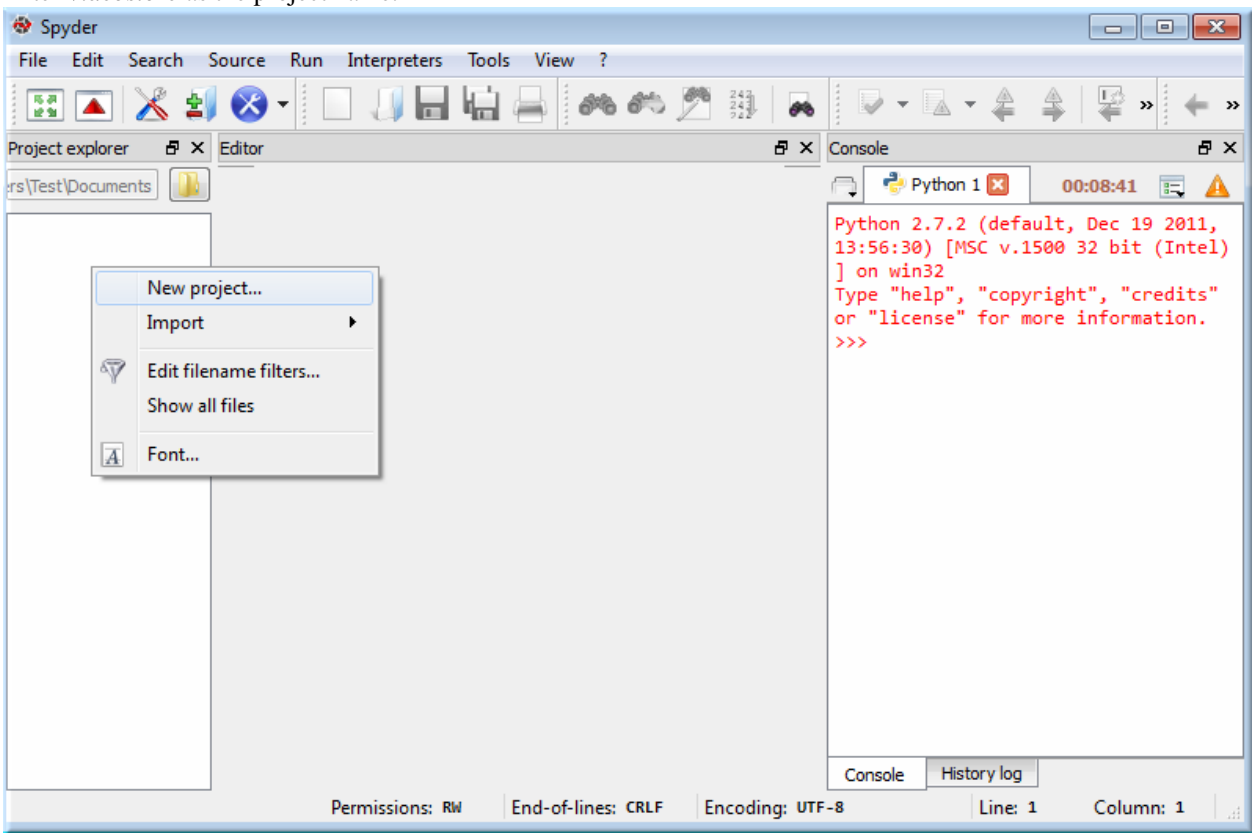
In the *Project Explorer* change the workspace directory, to the directory where you want to put your **Camelot** Projects.



Next, still in the *Project Explorer*, right click to create a new project using :

New Project

Enter *Videostore* as the project name.



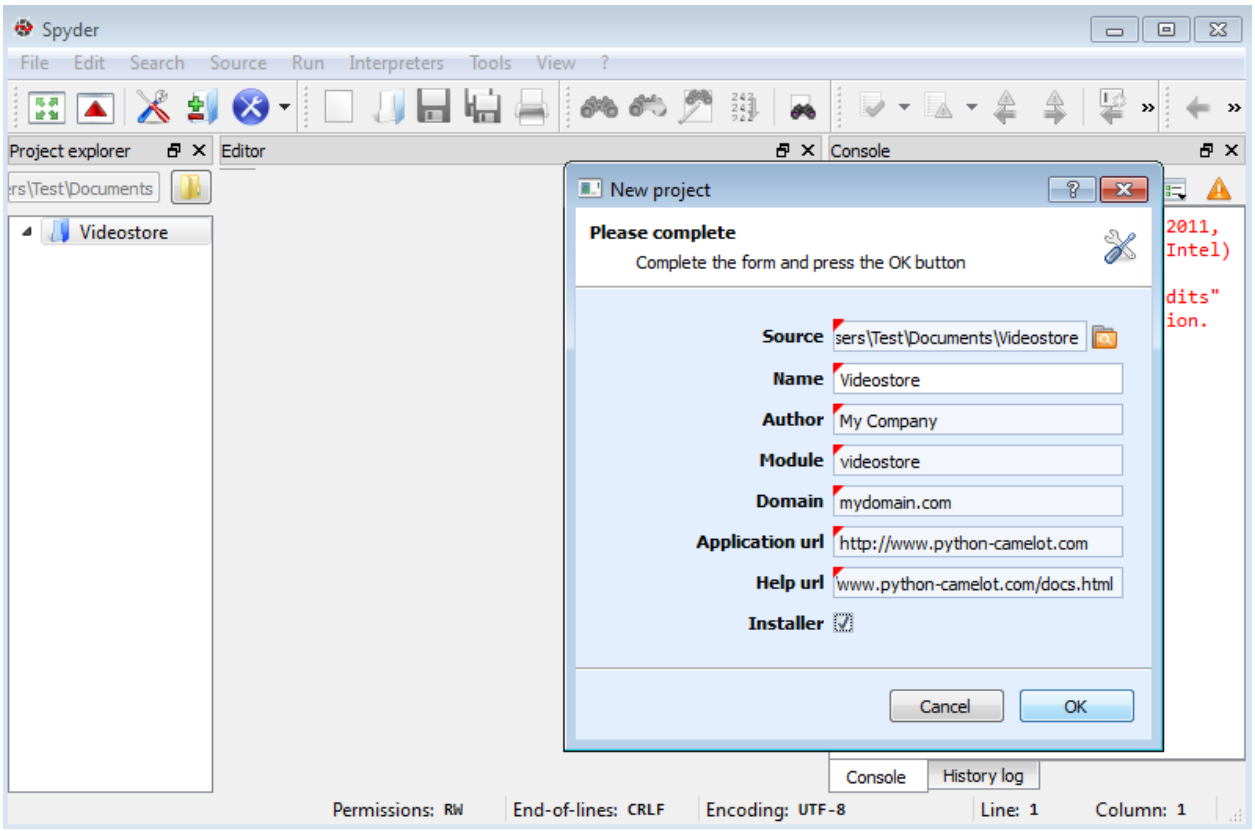
Starting a new Camelot project We begin with the creation of a new **Camelot** project, using the *camelot_admin* tool :

Start → All Programs → Python SDK → New Camelot Application

Note: From the command prompt (or shell), go to the directory in which the new project should be created. Type the following command:

```
python -m camelot.bin.camelot_admin
```

A dialog appears where the basic information of the application can be filled in. Select the newly created *Videostore* directory as the location of the source code.



Press *OK* to generate the source code of the project. The source code should now appear in the selected directory.

Main Window and Views To run the application, double click on the `main.py` file in **Spyder**, which contains the entry point of your **Camelot** application and run this file.

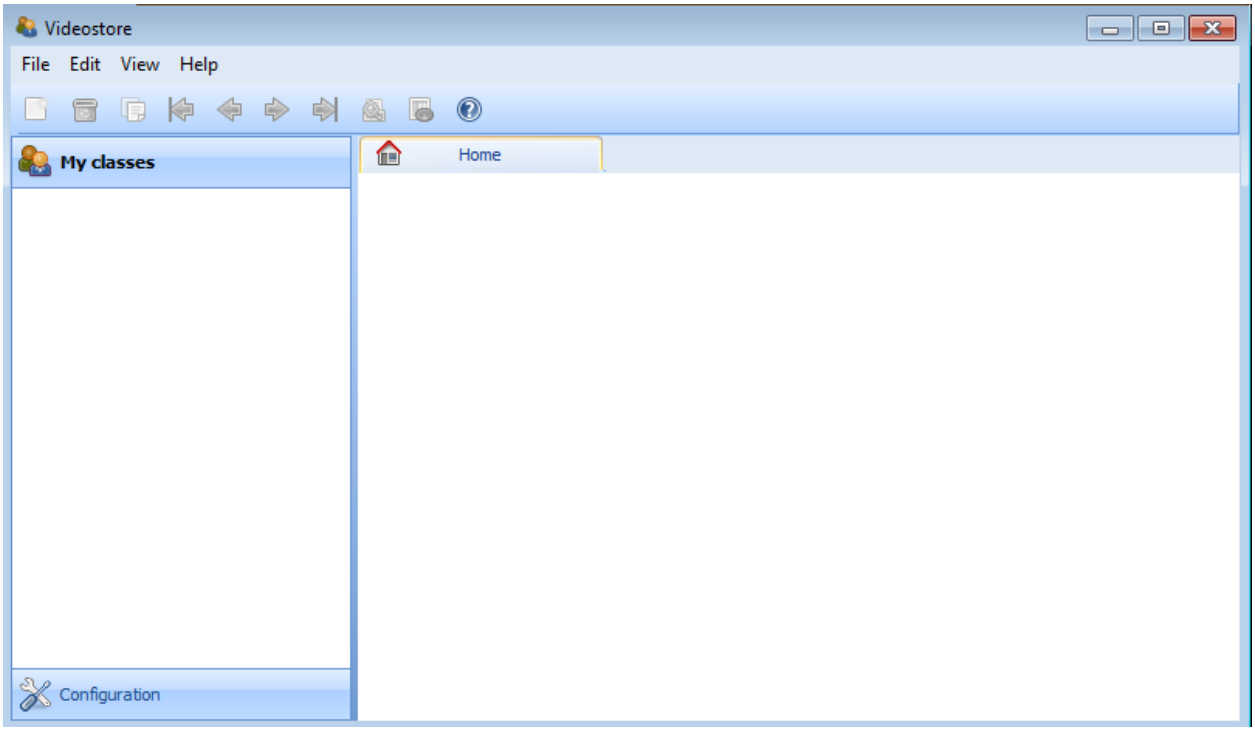
Run → *Run* → *Ok*

Note: From the command prompt, simply start the script

```
python main.py
```

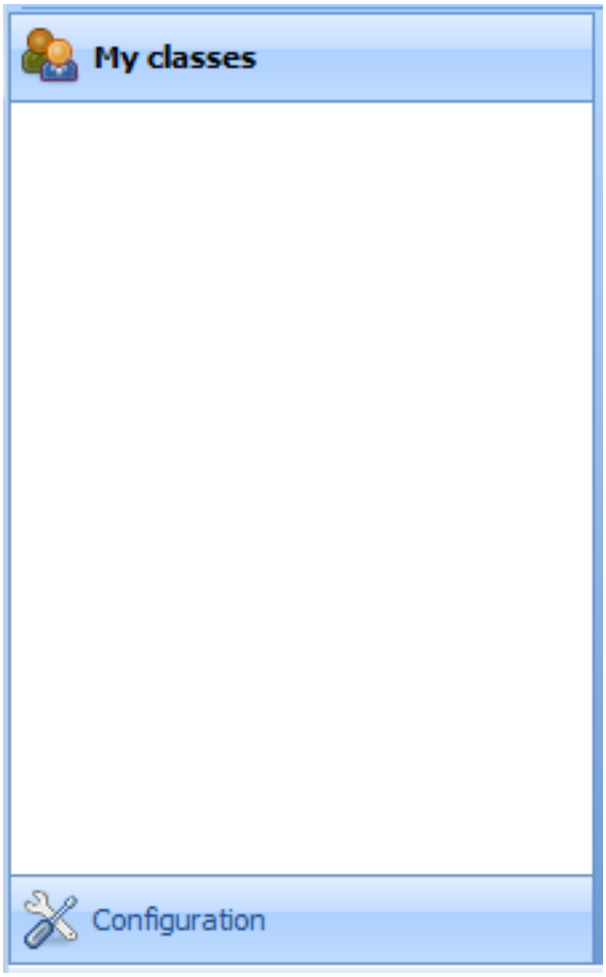
---

your Qt GUI should look like the one we show in the picture below:



The application has a customizable menu and toolbar, a left navigation pane, and a central area, where default the *Home* tab is opened, on which nothing is currently displayed.

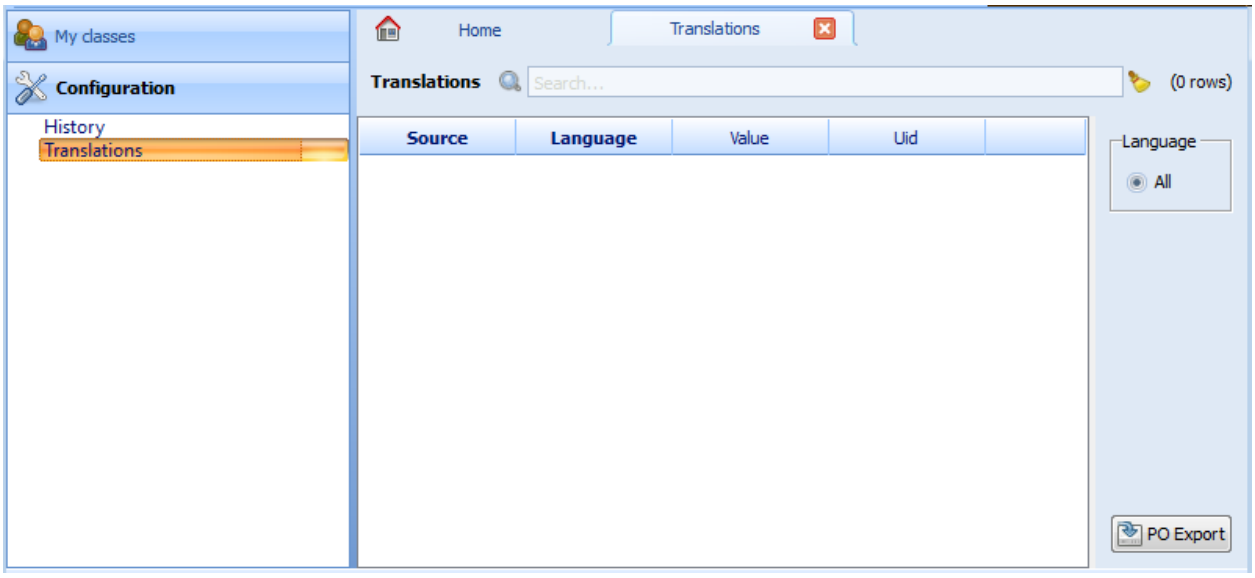
The navigation pane has its first *section* expanded.



The navigation pane uses *Sections* to group *Actions*. Each button in the navigation pane represents a *Section*, and each entry of the navigation tree is an *Action*. Most standard *Actions* open a single table view of an *Entity* in a new tab.

Notice that the application disables most of the menus and the toolbar buttons. When we open a table view, more options become available.

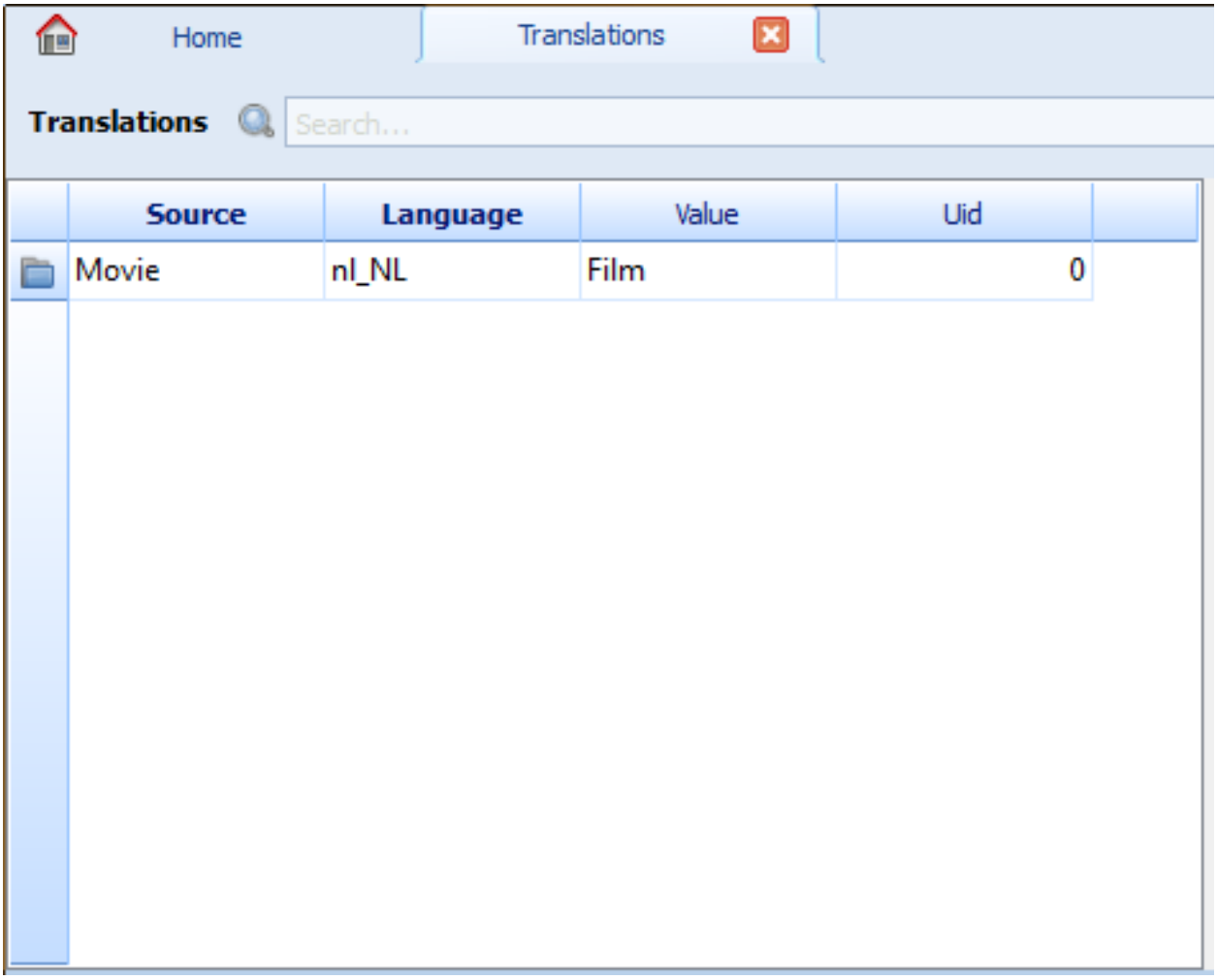
Entities are opened in the active tab, unless they are opened by selecting *Open in New Tab* from the context menu (right click) of the entity link, which will obviously open a new tab to right. Tabs can be closed by clicking the *X* in the tab itself.



Each row is a record with some fields that we can edit (others might not be editable). Let's now add a new row by clicking on the new icon (icon farthest the the left in the toolbar above the navigation pane).

We now see a new window, containing a form view with additional fields. Forms label **required** fields in bold.

Fill in a first and last name, and close the form. Camelot will automatically validate and echo the changes to the database. We can reopen the form by clicking on the blue folder icon in the first column of each row of the table. Notice also that there is now an entry in our table.



That's it for basic usages of the interface. Next we will write code for our database model.

Creating the Movie Model Let's first take a look at the `main.py` in our project directory. It contains a *my_settings* object which is appended to the global *settings*. The *Global settings* object contains the global configuration for things such as database and file location.

Now we can look at `model.py`. Camelot has already imported some classes for us. They are used to create our entities. Let's say we want a movie entity with a `title`, a short `description`, a `release date`, and a `genre`.

The aforementioned specifications translate into the following Python code, that we add to our `model.py` module:

```
from sqlalchemy import Unicode, Date
from sqlalchemy.schema import Column
from camelot.core.orm import Entity
from camelot.admin.entity_admin import EntityAdmin

class Movie( Entity ):

    __tablename__ = 'movie'

    title = Column( Unicode(60), nullable = False )
    short_description = Column( Unicode(512) )
    release_date = Column( Date() )
```

```
    genre = Column( Unicode(15) )
```

Note: The complete source code of this tutorial can be found in the `camelot_example` folder of the Camelot source code.

`Movie` inherits `camelot.core.orm.Entity`, which is the declarative base class for all objects that should be stored in the database. We use the `__tablename__` attribute to to name the table ourselves in which the data will be stored, otherwise a default tablename would have been used.

Our entity holds four fields that are stored in columns in the table.

```
title = Column( Unicode(60), nullable = False )
```

`title` holds up to 60 unicode characters, and cannot be left empty:

```
short_description = Column( Unicode(512) )
```

`short_description` can hold up to 512 characters:

```
release_date = Column( Date() )
genre = Column( Unicode(15) )
```

`release_date` holds a date, and `genre` up to 15 unicode characters:

For more information about defining models, refer to the SQLAlchemy Declarative extension.

The different SQLAlchemy column types used are described here. Finally, custom Camelot fields are documented in the section *camelot-column-types*.

Let's now create an `EntityAdmin` subclass

The EntityAdmin Subclass We have to tell Camelot about our entities, so they show up in the GUI. This is one of the purposes of `camelot.admin.entity_admin.EntityAdmin` subclasses. After adding the `EntityAdmin` subclass, our `Movie` class now looks like this:

```
class Movie( Entity ):

    __tablename__ = 'movie'

    title = Column( Unicode(60), nullable = False )
    short_description = Column( Unicode(512) )
    release_date = Column( Date() )
    genre = Column( Unicode(15) )

    def __unicode__( self ):
        return self.title or 'Untitled movie'

    class Admin( EntityAdmin ):
        verbose_name = 'Movie'
        list_display = ['title', 'short_description', 'release_date', 'genre']
```

We made `Admin` an inner class to strengthen the link between it and the `Entity` subclass. Camelot does not force us. Assign your `EntityAdmin` class to the `Admin` `Entity` member to put it somewhere else.

`verbose_name` will be the label used in navigation trees.

The last attribute is interesting; it holds a list containing the fields we have defined above. As the name suggests, `list_display` tells Camelot to only show the fields specified in the list. `list_display` fields are also taken as the default fields to show on a form.

In our case we want to display four fields: `title`, `short_description`, `release_date`, and `genre` (that is, all of them.)

The fields displayed on the form can optionally be specified too in the `form_display` attribute.

We also add a `__unicode__()` method that will return either the title of the movie entity or `'Untitled movie'` if title is empty. The `__unicode__()` method will be called in case Camelot needs a textual representation of an object, such as in a window title.

Let's move onto the last piece of the puzzle.

Configuring the Application We are now working with `application_admin.py`. One of the tasks of `application_admin.py` is to specify the sections in the left pane of the main window.

The created application has a class, `MyApplicationAdmin`. This class is a subclass of `camelot.admin.application_admin.ApplicationAdmin`, which is used to control the overall look and feel of every Camelot application.

To change sections in the left pane of the main window, simply overwrite the `get_sections` method, to return a list of the desired sections. By default this method contains:

```
def get_sections(self):
    from camelot.model.memento import Memento
    from camelot.model.i18n import Translation
    return [ Section( _('My classes'),
                      self,
                      Icon('tango/22x22/apps/system-users.png'),
                      items = [] ),
             Section( _('Configuration'),
                      self,
                      Icon('tango/22x22/categories/preferences-system.png'),
                      items = [Memento, Translation] )
           ]
```

which will display two buttons in the navigation pane, labelled `'My classes'` and `'Configurations'`, with the specified icon next to each label. And yes, the order matters.

We need to add a new section for our `Movie` entity, this is done by extending the list of sections returned by the `get_sections` method with a Movie section:

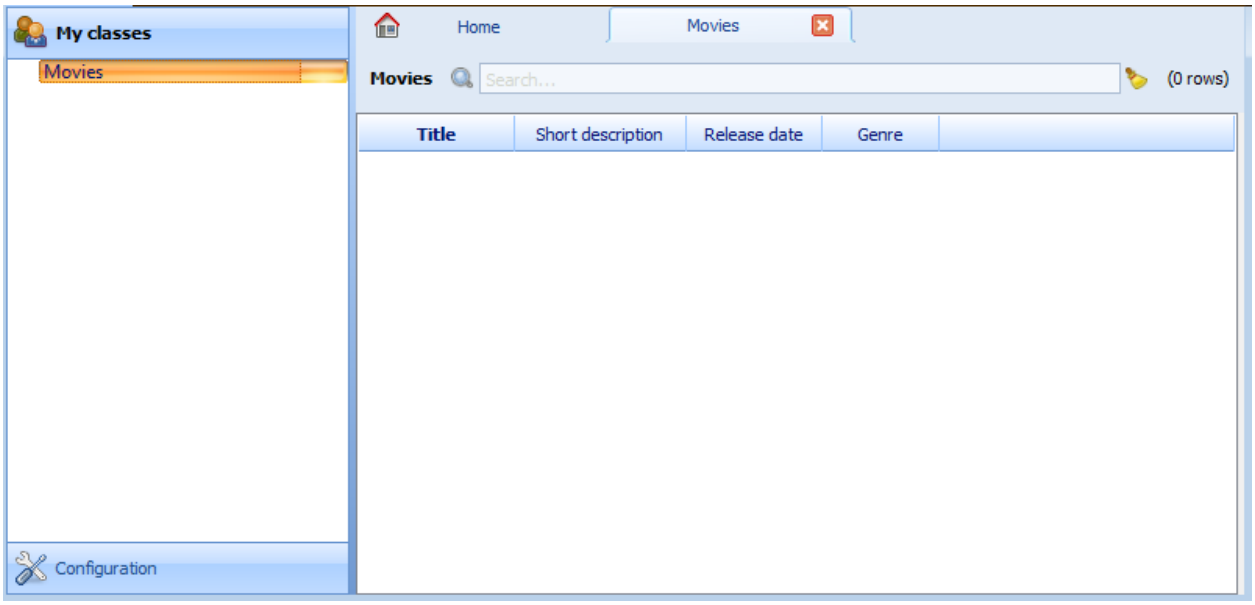
```
from videostore.model import Movie
return [ Section( _('Movie'),
                  self,
                  Icon('tango/22x22/apps/system-users.png'),
                  items = [Movie] ),
         Section( _('Configuration'),
                  self,
                  Icon('tango/22x22/categories/preferences-system.png'),
                  items = [Memento, Translation] )
       ]
```

The constructor of a section object takes the name of the section, a reference to the application admin object, the icon to be used and the items in the section. The items is a list of the entities for which a table view should shown.

Camelot comes with the Tango icon collection; we use a suitable icon for our movie section.

We can now try our application.

We see a new button the navigation pane labelled *'Movies'*. Clicking on it fills the navigation tree with the only entity in the movies's section. Clicking on this tree entry opens the table view. And if we click on the blue folder of each record, a form view appears as shown below.



That's it for the basics of defining an entity and setting it for display in Camelot. Next we look at relationships between entities.

Relationships We will be using SQLAlchemy's `sqlalchemy.orm.relationship` API. We'll relate a director to each movie. So first we need a `Director` entity. We define it as follows:

```
class Director( Entity ):

    __tablename__ = 'director'

    name = Column( Unicode( 60 ) )
```

Even if we define only the `name` column, Camelot adds an `id` column containing the primary key of the `Director` Entity. It does so because we did not define a primary key ourselves. This primary key is an integer number, unique for each row in the `director` table, and as such unique for each `Director` object.

Next, we add a reference to this primary key in the movie table, this is called the foreign key. This foreign key column, called `director_id` will be an integer number as well, with the added constraint that it can only contain values that are present in the `director` table its `id` column.

Because the `director_id` column is only an integer, we need to add the `director` attribute of type `relationship`. This will allow us to use the `director` property as a `Director` object related to a `Movie` object. The `relationship` attribute will find out about the `director_id` column and use it to attach a `Director` object to a `Movie` object

```
from sqlalchemy.schema import ForeignKey
from sqlalchemy.orm import relationship

class Movie( Entity ):

    __tablename__ = 'movie'

    title = Column( Unicode( 60 ),  nullable = False )
    short_description = Column( Unicode( 512 ) )
    release_date = Column( Date() )
    genre = Column( Unicode( 15 ) )
```

```
    director_id = Column( Integer, ForeignKey('director.id') )
    director = relationship( 'Director',
                             backref = 'movies' )

    class Admin( EntityAdmin ):
        verbose_name =  'Movie'
        list_display = [ 'title',
                         'short_description',
                         'release_date',
                         'genre',
                         'director' ]

    def __unicode__( self ):
        return self.title or 'untitled movie'
```

We also inserted `'director'` in `list_display`.

To be able to have the movies accessible from a director, a `backref` is defined in the *director* relationship. This will result in a `movies` attribute for each director, containing a list of movie objects.

Our `Director` entity needs an administration class as well. We will also add `__unicode__()` method as suggested above. The entity now looks as follows:

```
class Director( Entity ):
    __tablename__ = 'director'

    name = Column( Unicode(60) )

    class Admin( EntityAdmin ):
        verbose_name = 'Director'
        list_display = [ 'name' ]
        form_display = list_display + ['movies']

    def __unicode__(self):
        return self.name or 'unknown director'
```

Note: Whenever the model changes, the database needs to be updated. This can be done by hand, or by dropping and recreating the database (or deleting the sqlite file). By default Camelot stores the data in an local directory specified by the operating system. Look in the startup logs to see where they are stored on your system, look for a line like

```
[INFO    ] [camelot.core.conf] - store database and media in /home/username/.camelot/videostore
```

For completeness the two entities are once again listed below:

```
class Movie( Entity ):

    __tablename__ = 'movie'

    title = Column( Unicode( 60 ), nullable = False )
    short_description = Column( Unicode( 512 ) )
    release_date = Column( Date() )
    genre = Column( Unicode( 15 ) )

    director_id = Column( Integer, ForeignKey('director.id') )
    director = relationship( 'Director',
                             backref = 'movies' )

    class Admin( EntityAdmin ):
```

```
        verbose_name =  'Movie'
        list_display = [ 'title',
                         'short_description',
                         'release_date',
                         'genre',
                         'director' ]

    def __unicode__( self ):
        return self.title or 'untitled movie'

class Director( Entity ):
    __tablename__ = 'director'

    name = Column( Unicode(60) )

    class Admin( EntityAdmin ):
        verbose_name = 'Director'
        list_display = [ 'name' ]
        form_display = list_display + ['movies']

    def __unicode__(self):
        return self.name or 'unknown director'
```

The last step is to fix `application_admin.py` by adding the following lines to the Director entity to the Movie section:

```
Section( 'Movies',
         self,
         Icon( 'tango/22x22/mimetypes/x-office-presentation.png' ),
         items = [ Movie, Director ])
```

This takes care of the relationship between our two entities.

We have just learned the basics of Camelot, and have a nice movie database application we can play with. In another tutorial, we will learn more advanced features of Camelot.

Creating a Report with Camelot

With the Movie Database Application as our starting point, we're going to use the reporting framework in this tutorial. We will create a report of each movie, which we can access from the movie detail page.

Massaging the model First of all we need to create a button to access our report. This is easily done by specifying a form_action, right in the Admin subclass of the model. Our appended code will be:

```
form_actions = [MovieSummary()]
```

The action is described in the MovieSummary class, which we'll discuss next. Note that it needs to imported, obviously:

```
from movie_summary import MovieSummary
```

So the movie model admin will look like this:

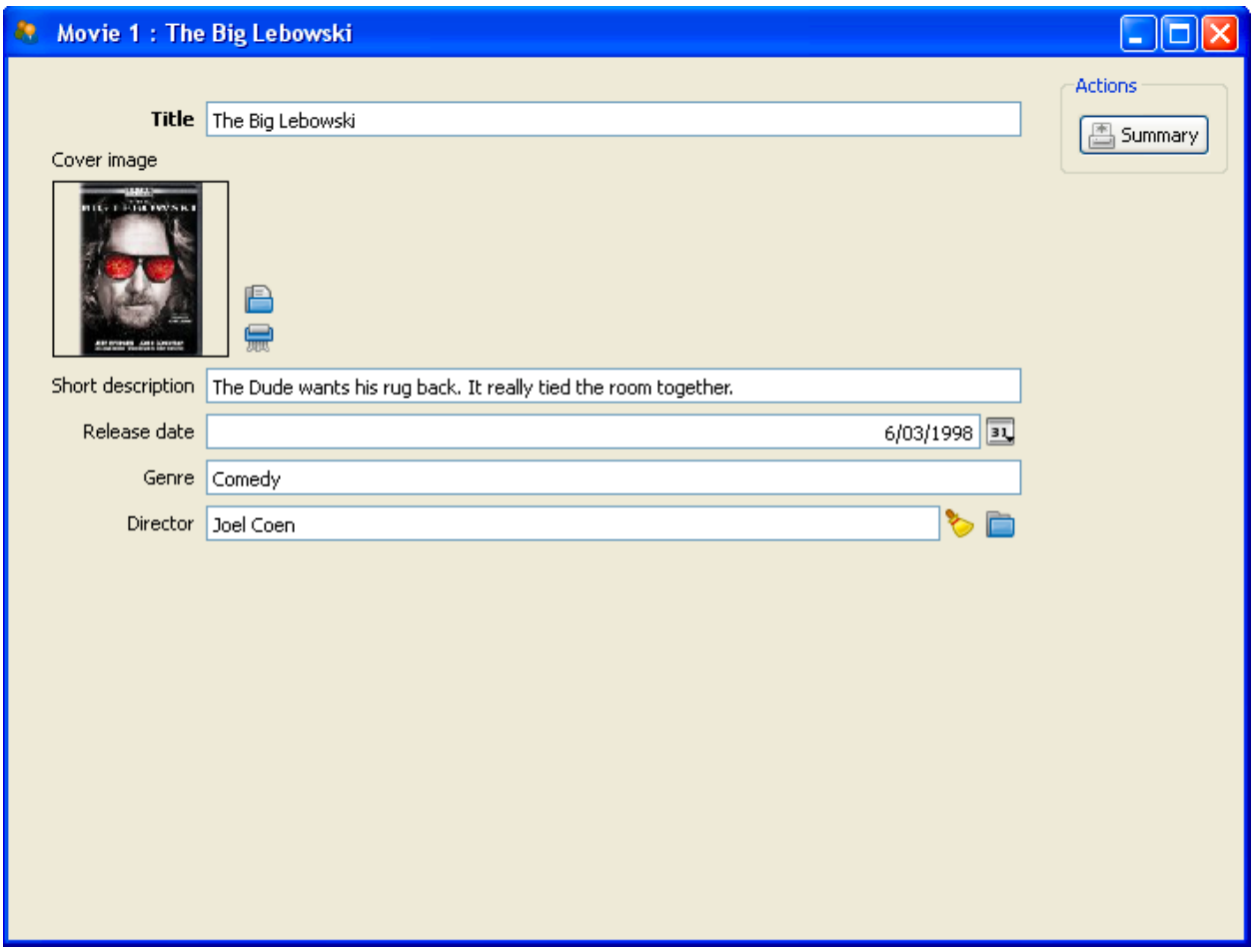
```
class Admin(EntityAdmin):
        from movie_summary import MovieSummary
        verbose_name = _('Movie')
        list_display = [
```

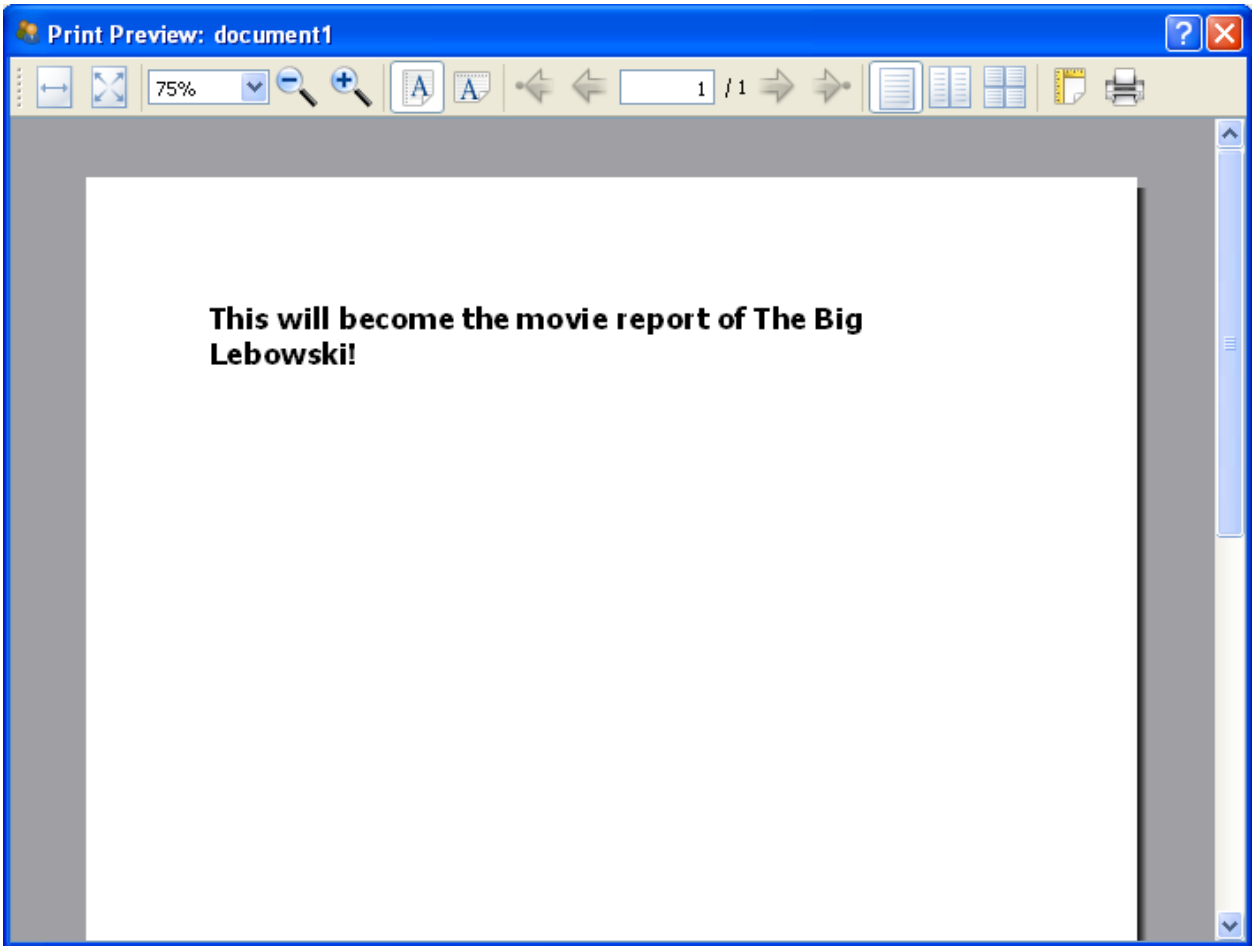
```
            'title',
            'short_description',
            'release_date',
            'genre',
            'director'
    ]
    form_display = [
            'title',
            'cover_image',
            'short_description',
            'release_date',
            'genre',
            'director'
    ]
    form_actions = [
            MovieSummary()
    ]
```

The Summary class In the MovieSummary class, which is a child class of `camelot.admin.action.base.Action`, we need to override just one method; the `model_run()` method, which has the *model_context* object as its argument. This makes accessing the *Movie* object very easy as we'll see in a minute. The *model_run* method will yield ..., have a guess.... Exactly, a print preview:

```
class MovieSummary( Action ):

    verbose_name = _('Summary')

    def model_run(self, model_context):
        from camelot.view.action_steps import PrintHtml
        movie = model_context.get_object()
        yield PrintHtml( "<h1>This will become the movie report of %s!</h1>" % movie.title )
```

You can already test this. You should see a button in the "Actions" section, on the right of the Movie detail page. Click this and a print preview should open with the text you let the html method return.


Movie 1 : The Big Lebowski
Actions
Summary
Title
The Big Lebowski
Cover image
Short description
The Dude wants his rug back. It really tied the room together.
Release date
6/03/1998
Genre
Comedy
Director
Joel Coen



Now let's make it a bit fancier.

Using Jinja templates Install and add Jinja2 to your PYTHONPATH. You can find it here: http://jinja.pocoo.org/2/ or at the cheeseshop http://pypi.python.org/pypi/Jinja2 . Now let's use its awesome powers.

First we'll make a base template. This will determine our look and feel for all the report pages. This is basically html and css with block definitions. Later we'll create the page movie summary template which will contain our model data. The movie summary template will inherit the base template, and provide content for the aforementioned blocks. The base template could look something like:

```
<html>
<head>
  <title>{% block page_head_title %}{% endblock %}</title>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  <style type="text/css">
body, html {
        font-family: Verdana, Arial, sans-serif;
}
{% block styles %}{% endblock %}
  </style>
</head>
<body>

<table id="page_header" width="100%">
  <tr>
```

```
    <td><h1>{% block page_header %}{% endblock %}</h1></td>
    <td align="right">{% block page_header_right %}{% endblock %}</td>
  </tr>

</table>
<hr>
<h2 id="page_title"><center>{% block page_title %}{% endblock %}</center></h2>
<hr>
{% block page_content %}{% endblock %}
<hr>
<div id="page_footer">{% block page_footer %}{% endblock %}</div>

</body>
</html>
```

We'll save this file as base.html in a directory called templates in our videostore. Like this base template, the movie summary template is html and css. Take a look at the example first:

```
{% extends 'base.html' %}
{% block styles %}{{ style }}{% endblock %}
{% block page_head_title %}{{ title }}{% endblock %}
{% block page_title %}{{ title }}{% endblock %}
{% block page_header %}{{ header }}{% endblock %}
{% block page_header_right %}
{% if cover %}
        <img src="{{ cover }}" alt="">
{% else %}
        (no cover)
{% endif %}
{% endblock %}
{% block page_content %}{{ content }}{% endblock %}
{% block page_footer %}{{ footer }}{% endblock %}
```

First we extend the base template, that way we don't need to worry about the boilerplate stuff, and keep our pages consistent, provided we create more reports of course. We can now fill in the blanks, erm blocks from the base template. We do that with placeholders which we'll define in the html method of our MovieSummary class. This way we can even add style to the page:

```
{% block styles %}{{ style }}{% endblock %}
```

We'll define this later. The templating language also allows basic flow control:

```
{% if cover %}
        <img src="{{ cover }}" alt="">
{% else %}
        (no cover)
{% endif %}
```

If there is no cover image, we'll show the string "(no cover)". We'll save this file as movie_summary.html in the templates directory.

Like i said earlier, we now need to define which values will go in the placeholders, so let's update our html method in the MovieSummary class. First, we import the needed elements:

```
import datetime
from jinja import Environment, FileSystemLoader
from pkg_resources import resource_filename
import videostore
from camelot.core.conf import settings
```

We'll be printing a date, so we'll need datetime. The Jinja classes to make use of our templates. And to locate our templates, we'll use the resource module, with our videostore. And load up the Jinja environment ...

```
fileloader = FileSystemLoader(resource_filename(videostore.__name__, 'templates'))
e = Environment(loader=fileloader)
```

Now we need to create a context dictionary to provide data to the templates. The keys of this dictionary are the placeholders we used in our movie_summary template, the values we can use from the model, which is passed as the o argument in the html method:

```
context = {
'header':o.title,
'title':'Movie Summary',
'style':'.label { font-weight:bold; }',
'content':'<span class="label">Description:</span> %s<br>\
        <span class="label">Release date:</span> %s<br>\
        <span class="label">Genre:</span> %s<br>\
        <span class="label">Director:</span> %s'
        % (o.short_description, o.release_date, o.genre, o.director),
'cover': os.path.join( settings.CAMELOT_MEDIA_ROOT(), 'covers', o.cover_image.name ),
'footer':'<br>copyright %s - Camelot' % datetime.datetime.now().year
}
```

Plain old Python dictionary. Check it out, we can even pass css in our setup.

Finally, we'll get the template from the Jinja environment and return the rendered result of our context:

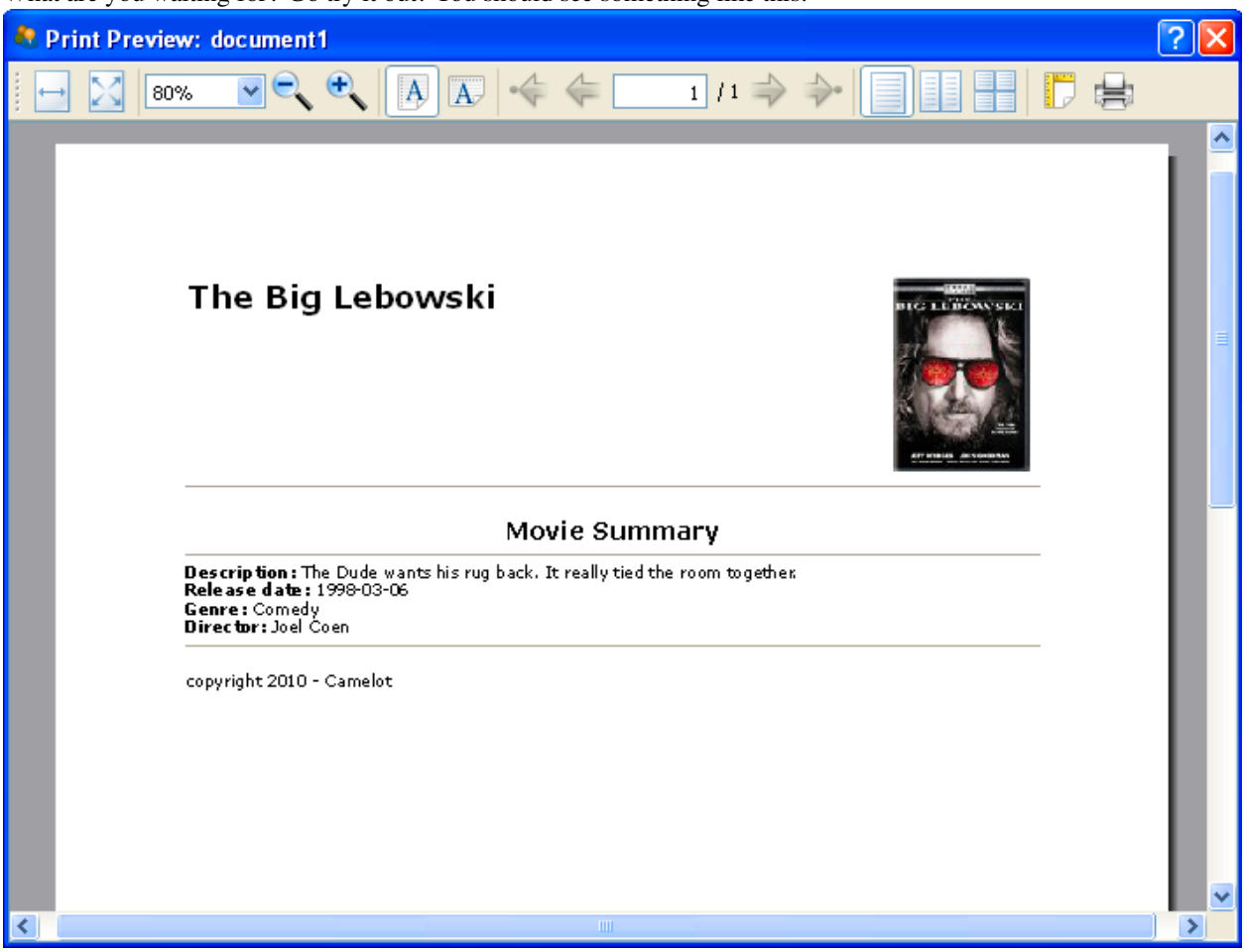
```
t = e.get_template('movie_summary.html')
return t.render(context)
```

So our finished method eventually looks like this:

```
from camelot.admin.action import Action

class MovieSummary( Action ):

    verbose_name = _('Summary')

    def model_run( self, model_context ):
        from camelot.view.action_steps import PrintHtml
        import datetime
        import os
        from jinja import Environment, FileSystemLoader
        from pkg_resources import resource_filename
        import videostore
        from camelot.core.conf import settings

        fileloader = FileSystemLoader(resource_filename(videostore.__name__, 'templates'))
        e = Environment(loader=fileloader)
        movie = model_context.get_object()
        context = {
                'header':movie.title,
                'title':'Movie Summary',
                'style':'.label { font-weight:bold; }',
                'content':'<span class="label">Description:</span> %s<br>\
                        <span class="label">Release date:</span> %s<br>\
                        <span class="label">Genre:</span> %s<br>\
                        <span class="label">Director:</span> %s'
                        % (movie.short_description, movie.release_date, movie.genre, movie.director),
                'cover': os.path.join( settings.CAMELOT_MEDIA_ROOT(), 'covers', movie.cover_image.nam
```

```
            'footer':'<br>copyright %s - Camelot' % datetime.datetime.now().year
        }
        t = e.get_template('movie_summary.html')
        yield PrintHtml( t.render(context) )
```

What are you waiting for? Go try it out! You should see something like this:



Add an import wizard to an application

In this tutorial we will add an import wizard to the movie database application created in the *Creating a Movie Database Application* tutorial.

We assume Camelot is properly *installed* and the movie database application is working.

_static/controls/main_window.png

Introduction Most applications need a way to import data. This data is often delivered in files generated by another application or company. To demonstrate this process we will build a wizard that allows the user to import cover images

into the movie database. For each image the user selects, a new Movie will be created with the selected image as a cover image.

Create an action All user interaction in Camelot is handled through *Actions*. For actions that run in the context of the application, we use the *Application Actions*. We first create a file `importer.py` in the same directory as `application_admin.py`.

In this file we create subclass of `camelot.admin.action.Action` which will be the entry point of the import wizard:

```
from camelot.admin.action import Action
from camelot.core.utils import ugettext_lazy as _

class ImportCovers( Action ):
    verbose_name = _('Import cover images')

    def model_run( self, model_context ):
        yield
```

So now we haven an `ImportCovers` action. Such an action has a `verbose_name` class attribute with the name of the action as shown to the user.

The most important method of the action is the `model_run` method, which will be triggered when the user clicks the action. This method should be a generator that yields an object whenever user interaction is required. Everything that happens inside the `model_run` method happens in a different thread than the GUI thread, so it will not block the GUI.

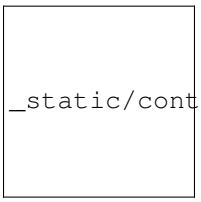
Add the action to the GUI Now the user needs to be able to trigger the action. We edit the `application_admin.py` file and make sure the `ImportCoversAction` is imported.

```
        from camelot_example.importer import ImportCovers
```

Then we add an instance of the `ImportCovers` action to the sections defined in the `get_sections` method of the `ApplicationAdmin`:

```
                Section( _('Movies'),
                         self,
                         Icon('tango/22x22/mimetypes/x-office-presentation.png'),
                         items = [ Movie,
                                   Tag,
                                   VisitorReport,
#                                   VisitorsPerDirector,
                                   ImportCovers() ]),
```

This will make sure the action pops up in the **Movies** section of the application.



_static/controls/navigation_pane.png

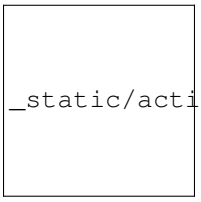
Select the files To make the action do something useful, we will implement its `model_run` method. Inside the `model_run` method, we can `yield` various `camelot.admin.action.base.ActionStep` objects to the

GUI. An `ActionStep` is a part of the action that requires user interaction (the user answering a question). The result of this interaction is returned by the `yield` statement.

To ask the user for a number of image files to import, we will pop up a file selection dialog inside the `model_run` method:

```
    def model_run( self, model_context ):
        from camelot.view.action_steps import ( SelectFile,
                                                UpdateProgress,
                                                Refresh,
                                                FlushSession )

        select_image_files = SelectFile( 'Image Files (*.png *.jpg);;All Files (*)' )
        select_image_files.single = False
        file_names = yield select_image_files
        file_count = len( file_names )
```

The `yield` statement returns a list of file names selected by the user.



Create new movies First make sure the `Movie` class has an `camelot.types.Image` field named `cover` which will store the image files.

```
    cover = Column( camelot.types.Image( upload_to = 'covers' ) )
```

Next we add to the `model_run` method the actual creation of new movies.

```
        import os
        from sqlalchemy import orm
        from camelot.core.orm import Session
        from camelot_example.model import Movie

        movie_mapper = orm.class_mapper( Movie )
        cover_property = movie_mapper.get_property( 'cover' )
        storage = cover_property.columns[0].type.storage
        session = Session()

        for i, file_name in enumerate(file_names):
            yield UpdateProgress( i, file_count )
            title = os.path.splitext( os.path.basename( file_name ) )[0]
            stored_file = storage.checkin( unicode( file_name ) )
            movie = Movie( title = unicode( title ) )
            movie.cover = stored_file

        yield FlushSession( session )
```

In this part of the code several things happen :

Store the images

In the first lines, we do some sqlalchemy magic to get access to the `storage` attribute of the `cover` field. This `storage` attribute is of type `camelot.core.files.storage.Storage`. The `Storage` represents the files managed by Camelot.

Create Movie objects

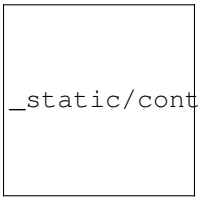
Then for each file, a new `Movie` object is created with as title the name of the file. For the `cover` attribute, the file is checked in into the `Storage`. This actually means the file is copied from its original directory to a directory managed by Camelot.

Write to the database

In the last line, the `session` is flushed and thus all changes are written to the database. The `camelot.view.action_steps.orm.FlushSession` action step flushes the session and propagetes the changes to the GUI.

Keep the user informed

For each movie imported, a `camelot.view.action_steps.update_progress.UpdateProgress` object is `yield` to the GUI to inform the user of the import progress. Each time such an object is yielded, the progress bar is updated.



Refresh the GUI The last step of the `model_run` method will be to refresh the GUI. So if the user has the `Movies` table open when importing, this table will show the newly created movies.

```
        yield Refresh()
```

Result This is how the resulting `importer.py` file looks like :

```
from camelot.admin.action import Action
from camelot.core.utils import ugettext_lazy as _
from camelot.view.art import Icon

class ImportCovers( Action ):
    verbose_name = _('Import cover images')
    icon = Icon('tango/22x22/mimetypes/image-x-generic.png')

# begin select files
    def model_run( self, model_context ):
        from camelot.view.action_steps import ( SelectFile,
                                                UpdateProgress,
                                                Refresh,
                                                FlushSession )

        select_image_files = SelectFile( 'Image Files (*.png *.jpg);;All Files (*)' )
        select_image_files.single = False
        file_names = yield select_image_files
        file_count = len( file_names )
# end select files
# begin create movies
        import os
        from sqlalchemy import orm
        from camelot.core.orm import Session
        from camelot_example.model import Movie
```

```
        movie_mapper = orm.class_mapper( Movie )
        cover_property = movie_mapper.get_property( 'cover' )
        storage = cover_property.columns[0].type.storage
        session = Session()

        for i, file_name in enumerate(file_names):
            yield UpdateProgress( i, file_count )
            title = os.path.splitext( os.path.basename( file_name ) )[0]
            stored_file = storage.checkin( unicode( file_name ) )
            movie = Movie( title = unicode( title ) )
            movie.cover = stored_file

        yield FlushSession( session )
# end create movies
# begin refresh
        yield Refresh()
# end refresh
```

Unit tests Once an action works, its important to keep it working as the development of the application continues. One of the advantages of working with generators for the user interaction, is that its easy to simulate the user interaction towards the `model_run()` method of the action. This is done by using the `send()` method of the generator that is returned when calling `model_run()` :

```
    def test_example_application_action( self ):
        from camelot_example.importer import ImportCovers
        from camelot_example.model import Movie
        # count the number of movies before the import
        movies = Movie.query.count()
        # create an import action
        action = ImportCovers()
        generator = action.model_run( None )
        select_file = generator.next()
        self.assertFalse( select_file.single )
        # pretend the user selected a file
        generator.send( [os.path.join( os.path.dirname(__file__), '..', 'camelot_example', 'media', '
        # continue the action till the end
        list( generator )
        # a movie should be inserted
        self.assertEqual( movies + 1, Movie.query.count() )
```

Conclusion We went through the basics of the action framework Camelot :

- Subclassing a `camelot.admin.action.Action` class
- Implementing the `model_run` method
- `yield` `camelot.admin.action.base.ActionStep` objects to interact with the user
- Add the `camelot.admin.action.base.Action` object to a `camelot.admin.section.Section` in the side pane

More `camelot.admin.action.base.ActionStep` classes can be found in the `camelot.view.action_steps` module.

Camelot Documentation

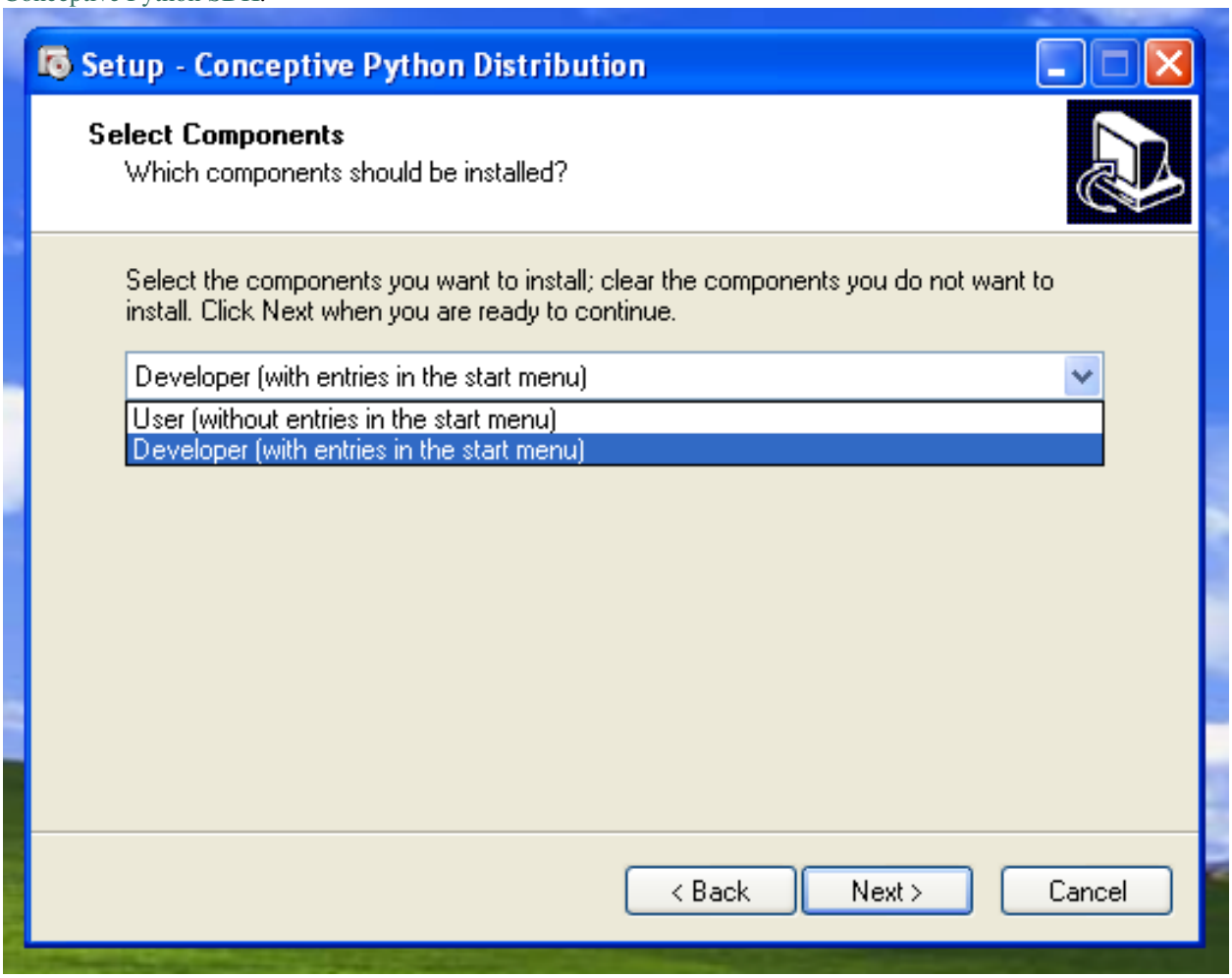
This is the reference documentation for developing projects using the Camelot library. The first time Camelot developer is encouraged to read *Creating models* and *Admin classes*.

The section *The Two Threads* is for developers whishing to maintain a responsive UI when faced with significant delays in their application code.

All other sections can be read on an as needed base.

Camelot Installation

All in one Windows installer When working on Windows, the easiest way to get up and running is through the Conceptive Python SDK.



This SDK is a Python distribution targeted at the development and deployment of QT based applications. This all in one installation of Camelot with all its dependencies is available in the shop.

From the Python Package Index First, make sure you have setup tools installed, Setup tools. If you are using a debian based distribution, you can type:

```
sudo apt-get install python-setuptools
```

Then use easy_install to install Camelot, under Linux this would be done by typing:

```
sudo easy_install camelot
```

Packages Linux distributions often offer packages for various applications, including Camelot and its dependencies :

- OpenSUSE build service.

From source When installing Camelot from source, you need to make sure all dependencies are installed and available in your **PYTHONPATH**.

Dependencies

In addition to PyQt 4.8 and Qt 4.8, Camelot needs these libraries :

SQLAlchemy==0.8.0 Jinja2==2.6 chardet==2.1.1 xlwt==0.7.4 xlrd==0.9.0

Releases

The source code of a release can be downloaded from the Python Package Index and then extracted:

```
tar xzvf Camelot-10.07.02.tar.gz
```

Repository

The latest and greatest version of the source can be checked out from the Bitbucket repository:

```
hg clone https://bitbucket.org/conceptive/camelot
```

Adapting PYTHONPATH

You need to make sure Camelot and all its dependencies are in the **PYTHONPATH** before you start using it.

Verifiy the installation To verify if you have Camelot installed and available in the **PYTHONPATH**, fire up a python interpreter:

```
python
```

and issue these commands:

```
>>> import camelot
>>> print camelot.__version__
>>> import sqlalchemy
>>> print sqlalchemy.__version__
>>> import PyQt4
```

None of them should raise an ImportError.

Creating models

Camelot makes it easy to create views for any type of *Python* objects.

SQLAlchemy is a very powerful Object Relational Mapper (ORM) with lots of possibilities for handling simple or sophisticated datastructures. The SQLAlchemy website has extensive documentation on all these features. An important part of Camelot is providing an easy way to create views for objects mapped through SQLAlchemy.

SQLAlchemy comes with the Declarative extension to make it easy to define an ORM mapping using the Active Record Pattern. This is used through the documentation and in the example code.

To use *Declarative*, threre are some base classes that should be imported:

```
from camelot.core.orm import Entity
from camelot.admin.entity_admin import EntityAdmin

from sqlalchemy import sql
from sqlalchemy.schema import Column
import sqlalchemy.types
```

Those are :

- `camelot.core.orm.Entity` is the declarative base class provided by Camelot for all classes that are mapped to the database, and is a subclass of `camelot.core.orm.entity.EntityBase`
- `camelot.admin.entity_admin.EntityAdmin` is the base class that describes how an *Entity* subclass should be represented in the GUI
- `sqlalchemy.schema.Column` describes a column in the database and a field in the model
- `sqlalchemy.types` contains the various column types that can be used

Next a model can be defined:

```
class Tag(Entity):

    __tablename__ = 'tags'

    name = Column( sqlalchemy.types.Unicode(60), nullable = False )
    movies = ManyToMany( 'Movie',
                         tablename = 'tags_movies__movies_tags',
                         local_colname = 'movies_id',
                         remote_colname = 'tags_id' )

    def __unicode__( self ):
        return self.name

    class Admin( EntityAdmin ):
        form_size = (400,200)
        list_display = ['name']

# begin visitor report definition
```

The code above defines the model for a *Tag* class, an object with only a name that can be related to other ojbects later on. This code has some things to notice :

- *Tag* is a subclass of `camelot.core.orm.Entity`,
- the `__tablename__` class attribute allows us to specify the name of the table in the database in which the tags will be stored.
- The `sqlalchemy.schema.Column` statement add fields of a certain type, in this case `sqlalchemy.types.Unicode`, to the *Tag* class as well as to the *tags* table
- The `__unicode__` method is implemented, this method will be called within Camelot whenever a textual representation of the object is needed, eg in a window title or a many to one widget. It's good practice to always implement the `__unicode__` method for all *Entity* subclasses.

When a new Camelot project is created, the *camelot-admin* tool creates an empty `models.py` file that can be used as a place to start the model definition.

Column types SQLAlchemy comes with a set of column types that can be used. These column types will trigger the use of a certain `QtGui.QDelegate` to visualize them in the views. Camelot extends those SQLAlchemy field types with some of its own.

An overview of field types from SQLAlchemy and Camelot is given in the table below :

All SQLAlchemy field types can be found in the `sqlalchemy.types` module. All additional Camelot field types can be found in the `camelot.types` module.

Relations SQLAlchemy uses the *relationship* function to define relations between classes. This function can be used within Camelot as well.

On top of this, Camelot provides some construct in the `camelot.core.orm.relationships` that make setting up relationships a bit easier.

Calculated Fields To display fields in forms that are not stored into the database but, are calculated at run time, two main options exist. Either those fields are calculated within the database or they are calculated by Python. Normal Python properties can be used to do the calculation in Python, whereas ColumnProperties can be used to do the logic in the database.

Python properties as fields Normal python properties can be used as fields on forms as well. In that case, there will be no introspection to find out how to display the property. Therefore the delegate (*Specifying delegates*) attribute should be specified explicitly.

```
import math

from camelot.admin.object_admin import ObjectAdmin
from camelot.view.controls import delegates

class Coordinate( object ):

  def __init__( self, x = 0, y = 0 ):
    self.id = 1
    self.x = x
    self.y = y

  @property
  def r( self ):
    return math.sqr( self.x**2, self.y**2 )

  class Admin( ObjectAdmin ):
    form_display = ['x', 'y', 'r']
    field_attributes = dict( x = dict( delegate = delegates.FloatDelegate,
                                       editable = True ),
                             y = dict( delegate = delegates.FloatDelegate,
                                       editable = True ),
                             r = dict( delegate = delegates.FloatDelegate ) )
```

By default, python properties are read-only. They have to be set to editable through the field attributes to make them writeable by the user.

Properties are also used to summarize information from multiple attributes and put them in a single field.

Cascading field changes Whenever the value of a field is changed, this change can cascade through the model by using properties to manipulate the field instead of manipulating it directly. The example below demonstrates how the value of y should be chopped when x is changed.

```
from camelot.admin.object_admin import ObjectAdmin
from camelot.view.controls import delegates

class Coordinate(object):

  def __init__(self):
    self.id = 1
    self.x = 0.0
    self.y = 0.0

  def _get_x(self):
    return self.x

  def _set_x(self, x):
    self.x = x
    self.y = max(self.y,x)

  _x = property(_get_x, _set_x)

  class Admin(ObjectAdmin):
    form_display = ['_x', 'y',]
    field_attributes = dict(_x=dict(delegate=delegates.FloatDelegate, name='x'),
                            y=dict(delegate=delegates.FloatDelegate),)
    form_size = (100,100)
```

Fields calculated by the database Having certain summary fields of your models filled by the database has the advantage that the heavy processing is moved from the client to the server. Moreover if the summary builds on information in related records, having the database build the summary reduces the need to transfer additional data from the database to the server.

To display fields in the table and the form view that are the result of a calculation done by the database, a `camelot.core.orm.properties.ColumnProperty` needs to be defined in the Declarative model. In this column property, the sql query can be defined using SQLAlchemy statements. In this example, the *Movie* class gains the *total_visitors* attribute which contains the sum of all visitors that went to a movie.

```
@ColumnProperty
def total_visitors( self ):
    return sql.select( [sql.func.sum( VisitorReport.visitors) ],
                                      VisitorReport.movie_id == self.id )
```

It's important to notice that the value of this field is calculated when the object is fetched from the database. When the user presses F9, all data in the application is refreshed from the database, and thus all column properties are recalculated.

Views Traditionally, in database land, **views** are queries defined at the database level that act like read-only tables. They allow reuse of common queries across an application, and are very suitable for reporting.

Using **SQLAlchemy** this traditional approach can be used, but a more dynamic approach is possible as well. We can map arbitrary queries to an object, and then visualize these objects with **Camelot**.

The model to start from

In the example movie project, we can take three parts of the model : Person, Movie and VisitorReport:

```
class Person( Party ):
    """Person represents natural persons
    """
    using_options( tablename = 'person' )
    party_id = Field( camelot.types.PrimaryKey(),
                      ForeignKey('party.id'),
                      primary_key = True )
    __mapper_args__ = {'polymorphic_identity': u'person'}
    first_name = Field( Unicode( 40 ), required = True )
    last_name = Field( Unicode( 40 ), required = True )
```

There is a relation between Person and Movie through the director attribute:

```
class Movie( Entity ):

    __tablename__ = 'movies'

    title = Column( sqlalchemy.types.Unicode(60), nullable = False )
    short_description = Column( sqlalchemy.types.Unicode(512) )
    releasedate = Column( sqlalchemy.types.Date )
    genre = Column( sqlalchemy.types.Unicode(15) )
    rating = Column( camelot.types.Rating() )
    #
    # All relation types are covered with their own editor
    #
    director = ManyToOne('Person')
    cast = OneToMany('Cast')
    visitor_reports = OneToMany('VisitorReport', cascade='delete')
    tags = ManyToMany('Tag',
                      tablename = 'tags_movies__movies_tags',
                      local_colname = 'tags_id',
                      remote_colname = 'movies_id' )
```

And a relation between Movie and VisitorReport:

```
class VisitorReport(Entity):

    __tablename__ = 'visitor_report'

    date = Column( sqlalchemy.types.Date,
                   nullable = False,
                   default = datetime.date.today )
    visitors = Column( sqlalchemy.types.Integer,
                       nullable = False,
                       default = 0 )
    movie = ManyToOne( 'Movie', required = True )
```

Definition of the view Suppose, we now want to display a table with the total numbers of visitors for all movies of a director.

We first define a plain old Python class that represents the expected results :

```
class VisitorsPerDirector(object):

    class Admin(EntityAdmin):
        verbose_name = _('Visitors per director')
        list_display = table.Table( [ table.ColumnGroup( _('Name and Visitors'), ['first_name', 'last
                                      table.ColumnGroup( _('Official'), ['birthdate', 'social_securit
                                    )
# end column group
```

Then define a function that maps the query that calculates those results to the plain old Python object :

```
def setup_views():
    from sqlalchemy.sql import select, func, and_
    from sqlalchemy.orm import mapper

    from camelot.model.party import Person
    from camelot_example.model import Movie, VisitorReport

    s = select([Person.party_id,
                Person.first_name.label('first_name'),
                Person.last_name.label('last_name'),
                Person.birthdate.label('birthdate'),
                Person.social_security_number.label('social_security_number'),
                Person.passport_number.label('passport_number'),
                func.sum( VisitorReport.visitors ).label('visitors'),],
                whereclause = and_( Person.party_id == Movie.director_party_id,
                                    Movie.id == VisitorReport.movie_id),
                group_by = [ Person.party_id,
                             Person.first_name,
                             Person.last_name,
                             Person.birthdate,
                             Person.social_security_number,
                             Person.passport_number, ] )

    s=s.alias('visitors_per_director')

    mapper( VisitorsPerDirector, s, always_refresh=True )
```

Put all this in a file called view.py

Put into action Then make sure the plain old Python object is mapped to the query, just after the Elixir model has been setup, by modifying the setup_model function in settings.py:

```
    def setup_model():
        from sqlalchemy.orm import configure_mappers
```

```
        from camelot.core.sql import metadata
        metadata.bind = settings.ENGINE()
        import camelot.model.party
        import camelot.model.authentication
        import camelot.model.i18n
        import camelot.model.fixture
        import camelot.model.memento
        import camelot.model.batch_job
        import camelot_example.model
        #
        # create the tables for all models, configure mappers first, to make
        # sure all deferred properties have been handled, as those could
        # create tables or columns
        #
        configure_mappers()
        metadata.create_all()
        from camelot.model.authentication import update_last_login
        #update_last_login()
        #
        # Load sample data with the fixure mechanism
        #
        from camelot_example.fixtures import load_movie_fixtures
        load_movie_fixtures()
        #
        # setup the views
        #
        from camelot_example.view import setup_views
        setup_views()
```

And add the plain old Python object to a section in the **ApplicationAdmin**:

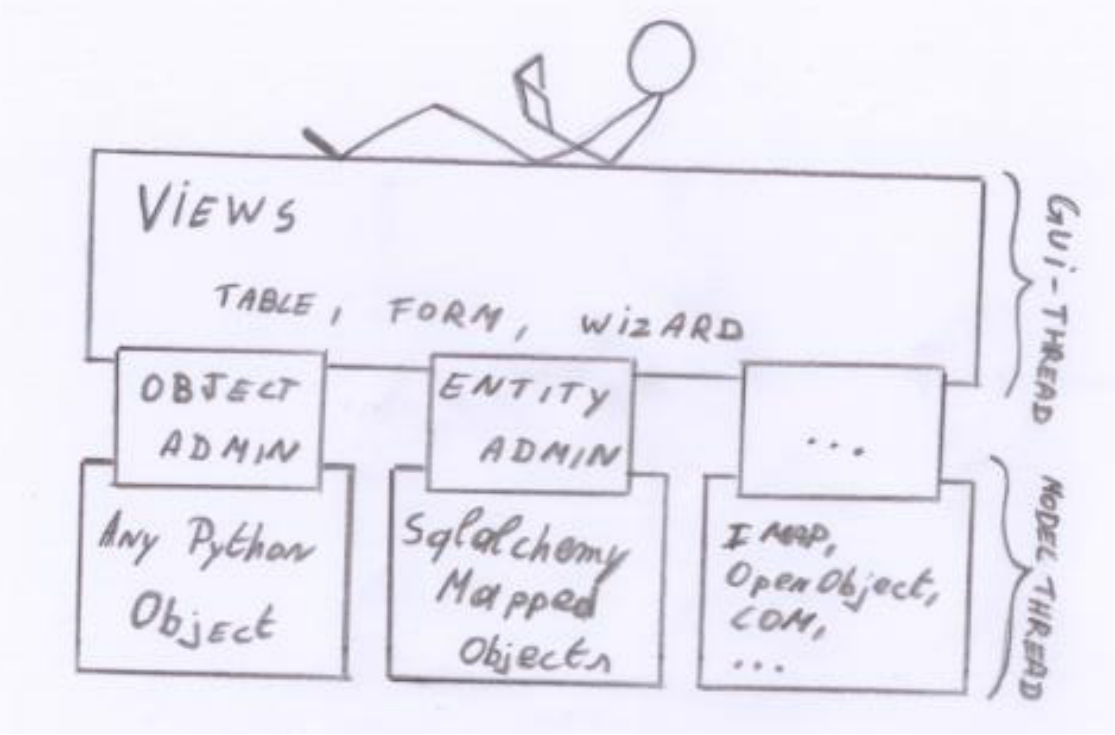
```
    def get_sections(self):

        from camelot.model.batch_job import BatchJob
        from camelot.model.memento import Memento
        from camelot.model.party import ( Person, Organization,
                                          PartyCategory )
        from camelot.model.i18n import Translation
        from camelot.model.batch_job import BatchJob, BatchJobType

        from camelot_example.model import Movie, Tag, VisitorReport
        from camelot_example.view import VisitorsPerDirector
# begin import action
        from camelot_example.importer import ImportCovers
# end import action

        return [
# begin section with action
                Section( _('Movies'),
                         self,
                         Icon('tango/22x22/mimetypes/x-office-presentation.png'),
                         items = [ Movie,
                                   Tag,
                                   VisitorReport,
#                                   VisitorsPerDirector,
                                   ImportCovers() ]),
# end section with action
                Section( _('Relation'),
```

```
                        self,
                        Icon('tango/22x22/apps/system-users.png'),
                        items = [ Person,
                                  Organization,
                                  PartyCategory ]),
                Section( _('Configuration'),
                        self,
                        Icon('tango/22x22/categories/preferences-system.png'),
                        items = [ Memento,
                                  Translation,
                                  BatchJobType,
                                  BatchJob
                                  ])
                ]
```

doc/../_static/entityviews/table_view_visitorsperdirector.png

Admin classes

The Admin classes are the classes that specify how objects should be visualized, they define the look, feel and behaviour of the Application. Most of the behaviour of the Admin classes can be tuned by changing their class attributes. This makes it easy to subclass a default Admin class and tune it to your needs.



ObjectAdmin Camelot is able to visualize any Python object, through the use of the `camelot.admin.object_admin.ObjectAdmin` class. However, subclasses exist that use introspection to facilitate the visualisation.

Each class that is visualized within Camelot has an associated Admin class which specifies how the object or a list of objects should be visualized.

Usually the Admin class is bound to the model class by defining it as an inner class of the model class:

```
class Options(object):
    """A python object in which we store the change in rating
    """

    def __init__(self):
        self.only_selected = True
        self.change = 1

    # Since Options is a plain old python object, we cannot
    # use an EntityAdmin, and should use the ObjectAdmin
    class Admin( ObjectAdmin ):
        verbose_name = _('Change rating options')
        form_display = ['change', 'only_selected']
        form_size = (100, 100)
        # Since there is no introspection, the delegate should
        # be specified explicitely, and set to editable
        field_attributes = {'only_selected':{'delegate':delegates.BoolDelegate,
                                             'editable':True},
                            'change':{'delegate':delegates.IntegerDelegate,
                                      'editable':True},
                            }

# begin change rating action definition
```

Most of the behaviour of the Admin class can be customized by changing the class attributes like *verbose_name*, *list_display* and *form_display*.

Other *Admin* classes can inherit *ObjectAdmin* if they want to provide additional functionallity, like introspection to set default field attributes.

EntityAdmin The `camelot.admin.entity_admin.EntityAdmin` class is a subclass of *ObjectAdmin* that can be used to visualize objects mapped to a database using SQLAlchemy.

The *EntityAdmin* uses introspection of the model to guess the default field attributes. This makes the definition of an *Admin* class less verbose.

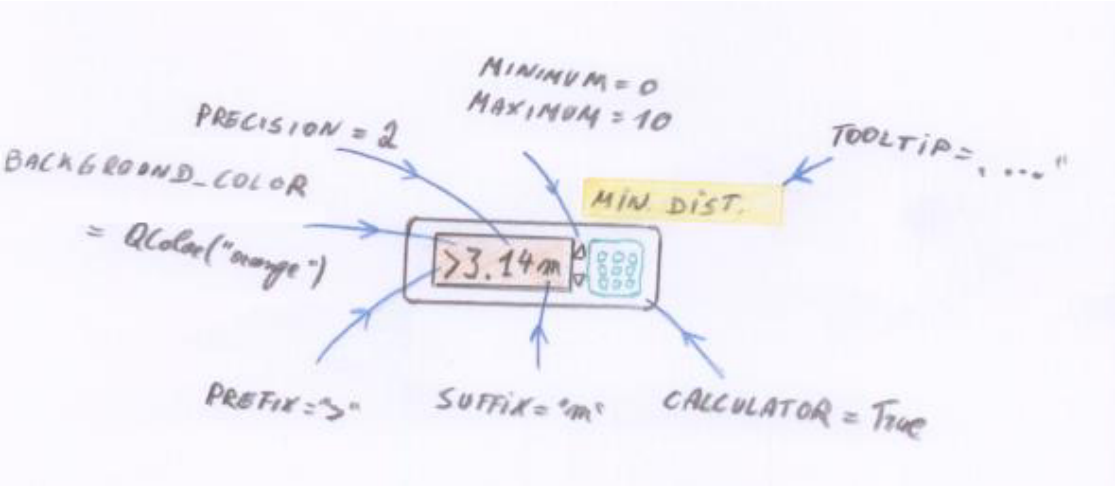
```
class Tag(Entity):

    __tablename__ = 'tags'

    name = Column( sqlalchemy.types.Unicode(60), nullable = False )
    movies = ManyToMany( 'Movie',
                         tablename = 'tags_movies__movies_tags',
                         local_colname = 'movies_id',
                         remote_colname = 'tags_id' )

    def __unicode__( self ):
        return self.name

    class Admin( EntityAdmin ):
```

```
        form_size = (400,200)
        list_display = ['name']

# begin visitor report definition
```

The `camelot.admin.entity_admin.EntityAdmin` provides some additonal attributes on top of those provided by `camelot.admin.object_admin.ObjectAdmin`, such as *list_filter* and *list_search*

Others

Field Attributes



Field attributes are the most convenient way to customize an application, they can be specified through the *field_attributes* dictionary of an *Admin* class :

```
class VisitorReport(Entity):

    __tablename__ = 'visitor_report'

    date = Column( sqlalchemy.types.Date,
                   nullable = False,
                   default = datetime.date.today )
    visitors = Column( sqlalchemy.types.Integer,
                       nullable = False,
                       default = 0 )
    movie = ManyToOne( 'Movie', required = True )
# end visitor report definition

    class Admin(EntityAdmin):
        verbose_name = _('Visitor Report')
        list_display = ['movie', 'date', 'visitors']
        field_attributes = {'visitors':{'minimum':0}}
```

Each combination of a delegate and an editor used to handle a field supports a different set of field attributes. To know which field attribute is supported by which editor or delegate, have a look at the *Delegates* documentation.

Static Field Attributes Static field attributes should be the same for every row in the same column, as such they should be specified as constant in the field attributes dictionary.

Dynamic Field Attributes Some field attributes, like background_color, can be dynamic. This means they can be specified as a function in the field attributes dictionary.

This function should take as its single argument the object on which the field attribute applies, as can be seen in the *background color example*

These are the field attributes that can be dynamic:

Overview of the field attributes

address_validator A function that verifies if a virtual address is valid, and eventually corrects it. The default implementation can is `camelot.view.controls.editors.virtualaddresseditor.default_address_validator()`

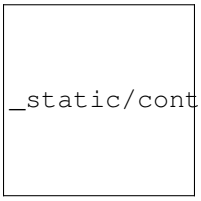
This function will be called while the user is editing the address, therefor it should take very little time to do the validation. If the address is invalid, this will be shown to the user, but it will not block the input of the address.

calculator `True` or `False` Indicates whether a calculator should be available when editing this field.

create_inline used in a one to many relation, if `False`, then a new entity will be created within a new window, if `True`, it will be created as a new line in the table.

column_width An integer forcing the column width of a field in a table view. The use of this field attribute is not recommended, since in most cases Camelot will figure out how wide a column should be. The use of *minimal_column_width* is advised to make sure a column has a certain width. But the *column_width* field attribute can be used to shrink the column width to arbitrary sizes, even if this might make the header unreadeable.

```
field_attributes = { 'first_name':{'column_width':8},
                     'suffix':{'column_width':8},}
```



directory `True` or `False` indicates if the file editor should point to a directory instead of a file. By default it points to a file.

editable `True` or `False`

Indicates whether the user can edit the field.

field_name This is the object name of the `QtGui.QWidget` that will be used as an editor for this field.

file_filter When the user is able to select a file or filename, use this filter to limit the available files.

length The maximum number of characters that can be entered in a text field.

minimum The minimum allowed value for `Integer` and `Float` delegates or their related delegates like the Star delegate.

maximum The maximum allowed value for `Integer` and `Float` delegates or their related delegates like the Star delegate.

precision The numerical precision that will be used to display `Float` values, this is unrelated to the precision in which they are stored.

choices A function taking as a single argument the object to which the field belongs. The function returns a list of tuples containing for each possible choice the value to be stored on the model and the value displayed to the user.

The use of `choices` forces the use of the ComboBox delegate:

```
field_attributes = {'state':{'choices':lambda o:[(1, 'Active'),
                                                 (2, 'Passive')]}}
```

minimal_column_width An integer specifying the minimal column width when this field is displayed in a table view. The width is expressed as the number of characters that should fit in the column:

```
field_attributes = {'name':{'minimal_column_width':50}}
```

will make the column wide enough to display at least 50 characters. The user will still be able to reduce the column size manually.

prefix String to display before a number

remove_original `True` or `False`

Set to `True` when a file should be deleted after it has been transfered to the storage.

single_step The size of a single step when the up and down arrows are used in on a float or an integer field.

suffix String to display after a number

tooltip A function taking as a single argument the object to which the field belongs. The function should return a string that will be used as a tooltip. The string may contain html markup.

```
from camelot.admin.object_admin import ObjectAdmin
from camelot.view.controls import delegates

def dynamic_tooltip_x(coordinate):
  return u'The <b>x</b> value of the coordinate, now set to %s'%(coordinate.x)

def dynamic_tooltip_y(coordinate):
  return u'The <b>y</b> value of the coordinate, now set to %s'%(coordinate.y)

class Coordinate(object):

  def __init__(self):
    self.id = 1
```

```
    self.x = 0.0
    self.y = 0.0

  class Admin(ObjectAdmin):
    form_display = ['x', 'y',]
    field_attributes = dict(x=dict(delegate=delegates.FloatDelegate,
                                   tooltip=dynamic_tooltip_x),
                            y=dict(delegate=delegates.FloatDelegate,
                                   tooltip=dynamic_tooltip_y),
                            )
    form_size = (100,100)
```

translate_content `True` or `False`

Wether the content of a field should be translated before displaying it. This only works for displaying content, not while editing it.

background_color A function taking as a single argument the object to which the field belongs. The function should return None if the default background should be used, or a QColor to be used as the background.

```
"""This Admin class turns the background of a Person's first
name pink if its first name doesn't start with a capital"""

from PyQt4.QtGui import QColor

from camelot.model.party import Person

def first_name_background_color(person):
    import string
    if person.first_name:
        if person.first_name[0] not in string.uppercase:
            return QColor('pink')

class Admin(Person.Admin):
    field_attributes = {'first_name':{'background_color':first_name_background_color}}
```

name The name of the field used, this defaults to the name of the attribute

target In case of relation fields, specifies the class that is at the other end of the relation. Defaults to the one found by introspection. This can be used to let a many2one editor always point to a subclass of the one found by introspection.

admin In case of relation fields, specifies the admin class that is to be used to visualize the other end of the relation. Defaults to the default admin class of the target class. This can be used to make the table view within a one2many widget look different from the default table view for the same object.

address_type Should be None or one of the Virtual Address Types, like 'phone' or 'email'. When specified, it indicates that a VirtualAddressEditor should only accept addresses of the specified type.

Customizing multiple field attributes When multiple field attributes need to be customized, specifying the *field_attributes* dictionary can become inefficient.

Several methods of the `camelot.admin.object_admin.ObjectAdmin` class can be overwritten to take care of this.

Instead of filling the *field_attributes* dictionary manually, the **:method:'camelot.admin.object_admin.ObjectAdmin.get_field_attribu** method can be overwritten :

When multiple dynamic field attributes need to execute the same logic to determine their value, it can be more efficient to overwrite the method **:method:'camelot.admin.object_admin.ObjectAdmin.get_dynamic_field_attributes'** and execute the logic once there and set the value for all dynamic field attributes at once.

The complement of *get_dynamic_field_attributes* is **:method:'camelot.admin.object_admin.ObjectAdmin.get_static_field_attribute**

Validators Before an object is written to the database it needs to be validated, and the user needs to be informed in case the object is not valid.

By default Camelot does some introspection on the model to check the validity of an object, to make sure it will be able to write the object to the database.

But this might not be enough. If more validation is needed, a custom Validator class can be defined. The default `camelot.admin.validator.entity_validator.EntityValidator` can be subclassed to create a custom validator. The new class should then be bound to the Admin class :

```
from camelot.admin.validator.entity_validator import EntityValidator
from camelot.admin.entity_admin import EntityAdmin

class PersonValidator(EntityValidator):

    def objectValidity(self, entity_instance):
        messages = super(PersonValidator,self).objectValidity(entity_instance)
        if (not entity_instance.first_name) or (len(entity_instance.first_name) < 3):
            messages.append("A person's first name should be at least 2 characters long")
        return messages

class Admin(EntityAdmin):
    verbose_name = 'Person'
    list_display = ['first_name', 'last_name']
    validator = PersonValidator
```

Its most important method is objectValidity, which takes an object as argument and should return a list of strings explaining why the object is invalid. These strings will then be presented to the user.

Notice that this method will always get called outside of the GUI thread, so the call will never block the GUI.

When the user tries to leave a form in an invalid state, a platform dependent dialog box will appear.

Customizing the Application

The **ApplicationAdmin** controls how the application behaves, it determines the sections in the left pane, the availability of help, the about box, the menu structure, etc.

The Application Admin Each Camelot application should subclass `camelot.admin.application_admin.ApplicationA` and overwrite some of its methods.

The look of the main window Most of these methods are based on the concept of *Actions*.

- `camelot.admin.application_admin.ApplicationAdmin.get_sections()`
- `camelot.admin.application_admin.ApplicationAdmin.get_actions()`
- `camelot.admin.application_admin.ApplicationAdmin.get_toolbar_actions()`
- `camelot.admin.application_admin.ApplicationAdmin.get_main_menu()`

Interaction with the Operating System

- `camelot.admin.application_admin.ApplicationAdmin.get_organization_name()`
- `camelot.admin.application_admin.ApplicationAdmin.get_organization_domain()`
- `camelot.admin.application_admin.ApplicationAdmin.get_name()`
- `camelot.admin.application_admin.ApplicationAdmin.get_version()`

The look of the application

- `camelot.admin.application_admin.ApplicationAdmin.get_splashscreen()`
- `camelot.admin.application_admin.ApplicationAdmin.get_stylesheet()`
- `camelot.admin.application_admin.ApplicationAdmin.get_translator()`
- `camelot.admin.application_admin.ApplicationAdmin.get_icon()`

The content of the help menu

- `camelot.admin.application_admin.ApplicationAdmin.get_about()`
- `camelot.admin.application_admin.ApplicationAdmin.get_help_url()`

Default behavior of the application

- `camelot.admin.application_admin.ApplicationAdmin.get_related_admin()`

The look of the form views

- camelot.admin.application_admin.ApplicationAdmin.get_related_toolbar_actions()
- camelot.admin.application_admin.ApplicationAdmin.get_form_actions()
- camelot.admin.application_admin.ApplicationAdmin.get_form_toolbar_actions()

Example

```
class MyApplicationAdmin(ApplicationAdmin):

    name = 'Camelot Video Store'

# begin sections
    def get_sections(self):

        from camelot.model.batch_job import BatchJob
        from camelot.model.memento import Memento
        from camelot.model.party import ( Person, Organization,
                                          PartyCategory )
        from camelot.model.i18n import Translation
        from camelot.model.batch_job import BatchJob, BatchJobType

        from camelot_example.model import Movie, Tag, VisitorReport
        from camelot_example.view import VisitorsPerDirector
# begin import action
        from camelot_example.importer import ImportCovers
# end import action

        return [
# begin section with action
                Section( _('Movies'),
                         self,
                         Icon('tango/22x22/mimetypes/x-office-presentation.png'),
                         items = [ Movie,
                                   Tag,
                                   VisitorReport,
#                                   VisitorsPerDirector,
                                   ImportCovers() ]),
# end section with action
                Section( _('Relation'),
                         self,
                         Icon('tango/22x22/apps/system-users.png'),
                         items = [ Person,
                                   Organization,
                                   PartyCategory ]),
                Section( _('Configuration'),
                         self,
                         Icon('tango/22x22/categories/preferences-system.png'),
                         items = [ Memento,
                                   Translation,
                                   BatchJobType,
                                   BatchJob
                                   ])
                ]
# end sections

# begin actions
    def get_actions(self):
```

```
        from camelot.admin.action import OpenNewView
        from camelot_example.model import Movie

        new_movie_action = OpenNewView( self.get_related_admin(Movie) )
        new_movie_action.icon = Icon('tango/22x22/mimetypes/x-office-presentation.png')

        return [new_movie_action]
# end actions
```

Example of a reduced application By reimplementing the default `get_sections()`, `get_main_menu()` and `get_toolbar_actions()`, it is possible to create a completely differently looking Camelot application.

_static/controls/reduced_main_window.png

```
    def get_toolbar_actions( self, toolbar_area ):
        from PyQt4.QtCore import Qt
        from camelot.model.party import Person
        from camelot.admin.action import application_action, list_action
        from model import Movie

        movies_action = application_action.OpenTableView( self.get_related_admin( Movie ) )
        movies_action.icon = Icon('tango/22x22/mimetypes/x-office-presentation.png')
        persons_action = application_action.OpenTableView( self.get_related_admin( Person ) )
        persons_action.icon = Icon('tango/22x22/apps/system-users.png')

        if toolbar_area == Qt.LeftToolBarArea:
            return [ movies_action,
                     persons_action,
                     list_action.OpenNewView(),
                     list_action.OpenFormView(),
                     list_action.DeleteSelection(),
                     application_action.Exit(),]

    def get_actions( self ):
        return []

    def get_sections( self ):
        return None

    def get_main_menu( self ):
        return None

    def get_stylesheet(self):
        from camelot.view import art
        return art.read('stylesheet/black.qss')
```

Creating Forms

This section describes how to place fields on forms and applying various layouts. It also covers how to customize forms to your specific needs. As with everything in Camelot, the goal of the framework is that you can create 80%

of your forms with minimal effort, while the framework should allow you to really customize the other 20% of your forms.

Form A form is a collection of fields organized within a layout. Each field is represented by its editor.

Usually forms are defined by specifying the *form_display* attribute of an Admin class :

```
from sqlalchemy.schema import Column
from sqlalchemy.types import Unicode, Date
from camelot.admin.entity_admin import EntityAdmin
from camelot.core.orm import Entity
from camelot.view import forms

class Movie( Entity ):
    title = Column( Unicode(60), nullable=False )
    short_description = Column( Unicode(512) )
    releasedate = Column( Date )

    class Admin(EntityAdmin):
        form_display = forms.Form( ['title', 'short_description', 'releasedate'] )
```

The *form_display* attribute should either be a list of fields to display or an instance of camelot.view.forms.Form or its subclasses.

Forms can be nested into each other :

```
from camelot.admin.entity_admin import EntityAdmin
from camelot.view import forms
from camelot.core.utils import ugettext_lazy as _

class Admin(EntityAdmin):
    verbose_name = _('person')
    verbose_name_plural = _('persons')
    list_display = ['first_name', 'last_name', ]
    form_display = forms.TabForm([('Basic', forms.Form(['first_name', 'last_name', 'contact_mechanis
                                  ('Official', forms.Form(['birthdate', 'social_security_number', 'pa
                                                           'passport_expiry_date','addresses',])), ])
```

Inheritance and Forms Just as Entities support inheritance, forms support inheritance as well. This avoids duplication of effort when designing and maintaining forms. Each of the Form subclasses has a set of methods to modify its content. In the example below a new tab is added to the form defined in the previous section.

```
from copy import deepcopy

from camelot.view import forms
from nested_form import Admin

class InheritedAdmin(Admin):
    form_display = deepcopy(Admin.form_display)
    form_display.add_tab('Work', forms.Form(['employers', 'directed_organizations', 'shares']))
```

Putting notes on forms

A note on a form is nothing more than a property with the NoteDelegate as its delegate and where the widget is inside a WidgetOnlyForm.

In the case of a Person, we display a note if another person with the same name already exists :

```
    def note(self):
        for person in self.__class__.query.filter_by(first_name=self.first_name, last_name=self.last_
            if person != self:
                return _('A person with the same name already exists')
```

Available Form Subclasses The `camelot.view.forms.Form` class has several subclasses that can be used to create various layouts. Those can be found in the `camelot.view.forms` module. Each subclass maps to a Qt Layout class.

Customizing Forms Several options exist for completely customizing the forms of an application.

Layout When the desired layout cannot be achieved with Camelot's form classes, a custom `camelot.view.forms.Form` subclass can be made to layout the widgets.

When subclassing the *Form* class, it's *render* method should be reimplemented to put the labels and the editors in a custom layout. The *render* method will be called by Camelot each time it needs the form. It should thus return a `QtGui.QWidget` to be used as the needed form.

The *render* method its first argument is the factory class `camelot.view.controls.formview.FormEditors`, through which editors and labels can be constructed. The editor widgets are bound to the data model.

```
from PyQt4 import QtGui

from camelot.view import forms
from camelot.admin.entity_admin import EntityAdmin
```

```
class CustomForm( forms.Form ):

    def __init__(self):
        super( CustomForm, self ).__init__(['first_name', 'last_name'])

    def render( self, editor_factory, parent = None, nomargins = False ):
        widget = QtGui.QWidget( parent )
        layout = QtGui.QFormLayout()
        layout.addRow( QtGui.QLabel('Please fill in the complete name :', widget ) )
        for field_name in self.get_fields():
            field_editor = editor_factory.create_editor( field_name, widget )
            field_label = editor_factory.create_label( field_name, field_editor, widget )
            layout.addRow( field_label, field_editor )
        widget.setLayout( layout )
        widget.setBackgroundRole( QtGui.QPalette.ToolTipBase )
        widget.setAutoFillBackground( True )
        return widget

class Admin(EntityAdmin):
    list_display = ['first_name', 'last_name']
    form_display = CustomForm()
    form_size = (300,100)
```

The form defined above puts the widgets into a `QtGui.QFormLayout` using a different background color, and adds some instructions for the user :

doc/../_static/form/custom_layout.png

Editors The editor of a specific field can be changed, by specifying an alternative `QtGui.QItemDelegate` for that field, using the *delegate* field attributes, see *Specifying delegates*.

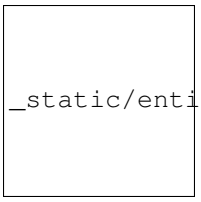
Tooltips Each field on the form can be given a dynamic tooltip, using the *tooltip* field attribute, see *tooltip*.

Buttons Buttons bound to a specific action can be put on a form, using the *form_actions* attribute, attribute of the Admin class : *Form Actions*.

Validation Validation is done at the object level. Before a form is closed validation of the bound object takes place, an invalid object will prevent closing the form. A custom validator can be defined : *Validators*

Actions

Introduction Besides displaying and editing data, every application needs the functions to manipulate data or create reports. In Camelot this is done through actions. Actions can appear as buttons on the side of a form or a table, as icons in a toolbar or as icons in the home workspace.



Every Action is build up with a set of Action Steps. An Action Step is a reusable part of an Action, such as for example, ask the user to select a file. Camelot comes with a set of standard Actions and Action Steps that are easily extended to manipulate data or create reports.

When defining Actions, a clear distinction should be made between things happening in the model thread (the manipulation or querying of data), and things happening in the gui thread (pop up windows or reports). The *The Two Threads* section gives more detail on this.

Summary In general, actions are defined by subclassing the standard Camelot `camelot.admin.action.Action` class

```
from camelot.admin.action import Action
from camelot.view.action_steps import PrintHtml
from camelot.core.utils import ugettext_lazy as _
from camelot.view.art import Icon

class PrintReport( Action ):

    verbose_name = _('Print Report')
    icon = Icon('tango/16x16/actions/document-print.png')
    tooltip = _('Print a report with all the movies')

    def model_run( self, model_context ):
        yield PrintHtml( 'Hello World' )
```

Each action has two methods, `gui_run()` and `model_run()`, one of them should be reimplemented in the subclass to either run the action in the gui thread or to run the action in the model thread. The default `Action.gui_run()` behavior is to pop-up a `ProgressDialog` dialog and start the `model_run()` method in the model thread.

`model_run()` in itself is a generator, that can yield `ActionStep` objects back to the gui, such as a `PrintHtml`.

The action objects can than be used a an element of the actions list returned by the `ApplicationAdmin.get_actions()` method:

```
    def get_actions(self):
        from camelot.admin.action import OpenNewView
        from camelot_example.model import Movie

        new_movie_action = OpenNewView( self.get_related_admin(Movie) )
        new_movie_action.icon = Icon('tango/22x22/mimetypes/x-office-presentation.png')

        return [new_movie_action]
```

or be used in the `ObjectAdmin.list_actions` or `ObjectAdmin.form_actions` attributes.

The *Add an import wizard to an application* tutorial has a complete example of creating and using and action.

What can happen inside `model_run()`

`yield` events to the GUI Actions need to be able to send their results back to the user, or ask the user for additional information. This is done with the `yield` statement.

Through `yield`, an Action Step is send to the GUI thread, where it creates user interaction, and sends it result back to the 'model_thread'. The model_thread will be blocked while the action in the GUI thread takes place, eg

```
yield PrintHtml( 'Hello World' )
```

Will pop up a print preview dialog in the GUI, and the model_run method will only continue when this dialog is closed.

Events that can be yielded to the GUI should be of type `camelot.admin.action.base.ActionStep`. Action steps are reusable parts of an action. Possible Action Steps that can be yielded to the GUI include:

- `camelot.view.action_steps.change_object.ChangeObject`
- `camelot.view.action_steps.change_object.ChangeObjects`
- `camelot.view.action_steps.print_preview.PrintChart`
- `camelot.view.action_steps.print_preview.PrintPreview`
- `camelot.view.action_steps.print_preview.PrintHtml`
- `camelot.view.action_steps.print_preview.PrintJinjaTemplate`
- `camelot.view.action_steps.open_file.OpenFile`
- `camelot.view.action_steps.open_file.OpenStream`
- `camelot.view.action_steps.open_file.OpenJinjaTemplate`
- `camelot.view.action_steps.gui.CloseView`
- `camelot.view.action_steps.gui.MessageBox`
- `camelot.view.action_steps.gui.Refresh`
- `camelot.view.action_steps.gui.OpenFormView`
- `camelot.view.action_steps.gui.ShowPixmap`
- `camelot.view.action_steps.gui.ShowChart`
- `camelot.view.action_steps.select_file.SelectFile`
- `camelot.view.action_steps.select_object.SelectObject`

keep the user informed about progress An `camelot.view.action_steps.update_progress.UpdateProgress` object can be yielded, to update the state of the progress dialog:

This should be done regulary to keep the user informed about the progres of the action:

```
movie_count = Movie.query.count()

report = '<table>'
for i, movie in enumerate( Movie.query.all() ):
    report += '<tr><td>%s</td></tr>'%(movie.name)
    yield UpdateProgress( i, movie_count )
report += '</table>'

yield PrintHtml( report )
```

Should the user have pressed the *Cancel* button in the progress dialog, the next yield of an UpdateProgress object will raise a `camelot.core.exception.CancelRequest`.

manipulation of the model The most important purpose of an action is to query or manipulate the model, all such things can be done in the `model_run()` method, such as executing queries, manipulating files, etc.

Whenever a part of the model has been changed, it might be needed to inform the GUI about this, so that it can update itself, the easy way of doing so is by yielding an instance of `camelot.view.action_steps.orm.FlushSession` such as:

```
movie.rating = 5
yield FlushSession( model_context.session )
```

This will flush the session to the database, and at the same time update the GUI so that the flushed changes are shown to the user by updating the visualisation of the changed movie on every screen in the application that displays this object. Alternative updates that can be generated are :

- `camelot.view.action_steps.orm.UpdateObject`, if one wants to inform the GUI an object has been updated.
- `camelot.view.action_steps.orm.DeleteObject`, if one wants to inform the GUI an object is going to be deleted.
- `camelot.view.action_steps.orm.CreateObject`, if one wants to inform the GUI an object has been created.

raise exceptions When an action fails, a normal Python `Exception` can be raised, which will pop-up an exception dialog to the user that displays a stack trace of the exception. In case no stack trace should be shown to the user, a `camelot.core.exception.UserException` should be raised. This will popup a friendly dialog :

_static/controls/user_exception.png

When the `model_run()` method raises a `camelot.core.exception.CancelRequest`, a `GeneratorExit` or a `StopIteration` exception, these are ignored and nothing will be shown to the user.

handle exceptions In case an unexpected event occurs in the GUI, a `yield` statement will raise a `camelot.core.exception.GuiException`. This exception will propagate through the action an will be ignored unless handled by the developer.

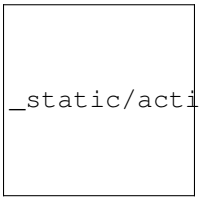
request information from the user The pop-up of a dialog that presents the user with a number of options can be triggered from within the `model_run()` method. This happens by transferring an **options** object back and forth between the **model_thread** and the **gui_thread**. To transfer such an object, this object first needs to be defined:

```
class Options( object ):

    def __init__(self):
        self.earliest_releasedate = datetime.date(2000, 1, 1)
        self.latest_releasedate = datetime.date.today()

    class Admin( ObjectAdmin ):
        form_display = [ 'earliest_releasedate', 'latest_releasedate' ]
        field_attributes = { 'earliest_releasedate':{'delegate':delegates.DateDelegate},
                             'latest_releasedate':{'delegate':delegates.DateDelegate}, }
```

Than a camelot.view.action_steps.change_object.ChangeObject action step can be yield to present the options to the user and get the filled in values back :

```
from PyQt4 import QtGui
from camelot.view import action_steps
options = NewProjectOptions()
yield action_steps.UpdateProgress( text = 'Request information' )
yield action_steps.ChangeObject( options )
```

Will show a dialog to modify the object:



When the user presses *Cancel* button of the dialog, the yield statement will raise a camelot.core.exception.CancelRequest.

Other ways of requesting information are :

- camelot.view.action_steps.select_file.SelectFile, to request to select an existing file to process or a new file to save information.

Issue SQLAlchemy statements Camelot itself only manipulates the database through objects of the ORM for the sake of make no difference between objects mapped to the database and plain old python objects. But for performance reasons, it is often desired to do manipulations directly through SQLAlchemy ORM or Core queries :

```
model_context.session.query( BatchJobType ).update( values = {'name':'accounting audit'},
                                                    synchronize_session = 'evaluate' )
```

States and Modes

States The widget that is used to trigger an action can be in different states. A camelot.admin.action.base.State object is returned by the camelot.admin.action.base.Action.get_state method. Subclasses of Action can reimplement this method to change the State of an action button.

This allows to hide or disable the action button, depending on the objects selected or the current object being displayed.

Modes An action widget can be triggered in different modes, for example a print button can be triggered as *Print* or *Export to PDF*. The different modes of an action are specified as a list of camelot.admin.action.base.Mode objects.

To change the modes of an Action, either specify the modes attribute of an Action or specify the modes attribute of the State returned by the Action.get_state() method.

Action Context Depending on where an action was triggered, a different context will be available during its execution in camelot.admin.action.base.Action.gui_run() and camelot.admin.action.base.Action.model_run().

The minimal context available in the *GUI thread* when gui_run() is called :

While the minimal contact available in the *Model thread* when model_run() is called :

Application Actions To enable Application Actions for a certain `ApplicationAdmin` overwrite its `ApplicationAdmin.get_actions()` method:

```
from camelot.admin.application_admin import ApplicationAdmin
from camelot.admin.action import Action

class GenerateReports( Action ):

    verbose_name = _('Generate Reports')

    def model_run( self, model_context):
        for i in range(10):
            yield UpdateProgress(i, 10)

class MyApplicationAdmin( ApplicationAdmin )

    def get_actions( self ):
        return [GenerateReports(),]
```

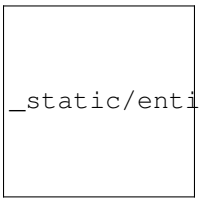
An action specified here will receive an `ApplicationActionGuiContext` object as the *gui_context* argument of the the `gui_run()` method, and a `ApplicationActionModelContext` object as the *model_context* argument of the `model_run()` method.

Form Actions A form action has access to the object currently visible on the form.

```
class BurnToDisk( Action ):

    verbose_name = _('Burn to disk')

    def model_run( self, model_context ):
        yield action_steps.UpdateProgress( 0, 3, _('Formatting disk') )
        time.sleep( 0.7 )
        yield action_steps.UpdateProgress( 1, 3, _('Burning movie') )
        time.sleep( 0.7 )
        yield action_steps.UpdateProgress( 2, 3, _('Finishing') )
        time.sleep( 0.5 )
```

To enable Form Actions for a certain `ObjectAdmin` or `EntityAdmin`, specify the `form_actions` attribute.

```
        #
        # create a list of actions available for the user on the form view
        #
        form_actions = [BurnToDisk()]
```



An action specified here will receive a `FormActionGuiContext` object as the *gui_context* argument of the `gui_run()` method, and a `FormActionModelContext` object as the *model_context* argument of the `model_run()` method.

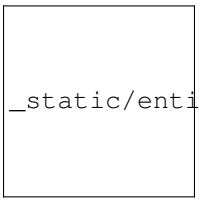
List Actions A list action has access to both all the rows displayed in the table (called the collection) and the rows selected by the user (called the selection) :

```
class ChangeRatingAction( Action ):
    """Action to print a list of movies"""

    verbose_name = _('Change Rating')

    def model_run( self, model_context ):
        #
        # the model_run generator method yields various ActionSteps
        #
        options = Options()
        yield ChangeObject( options )
        if options.only_selected:
            iterator = model_context.get_selection()
        else:
            iterator = model_context.get_collection()
        for movie in iterator:
            yield UpdateProgress( text = u'Change %s'%unicode( movie ) )
            movie.rating = min( 5, max( 0, (movie.rating or 0 ) + options.change ) )
        #
        # FlushSession will write the changes to the database and inform
        # the GUI
        #
        yield FlushSession( model_context.session )
```

To enable List Actions for a certain `ObjectAdmin` or `EntityAdmin`, specify the `list_actions` attribute:

```
        #
        # the action buttons that should be available in the list view
        #
        list_actions = [ChangeRatingAction()]
```

This will result in a button being displayed on the table view.



An action specified here will receive a `ListActionGuiContext` object as the *gui_context* argument of th the `gui_run()` method, and a `ListActionModelContext` object as the *model_context* argument of the `model_run()` method.

Reusing List and Form actions There is no need to define a different action subclass for form and list actions, as both their model_context have a **get_selection** method, a single action can be used both for the list and the form.

Available actions Camelot has a set of available actions that combine the various `ActionStep` subclasses. Those actions can be used directly or as an inspiration to build new actions:

- `camelot.admin.action.application_action.OpenNewView`
- `camelot.admin.action.application_action.OpenTableView`
- `camelot.admin.action.application_action.ShowHelp`
- `camelot.admin.action.application_action.ShowAbout`

- `camelot.admin.action.application_action.Backup`
- `camelot.admin.action.application_action.Restore`
- `camelot.admin.action.application_action.Refresh`
- `camelot.admin.action.form_action.CloseForm`
- `camelot.admin.action.list_action.CallMethod`
- `camelot.admin.action.list_action.OpenFormView`
- `camelot.admin.action.list_action.OpenNewView`
- `camelot.admin.action.list_action.ToPreviousRow`
- `camelot.admin.action.list_action.ToNextRow`
- `camelot.admin.action.list_action.ToFirstRow`
- `camelot.admin.action.list_action.ToLastRow`
- `camelot.admin.action.list_action.ExportSpreadsheet`
- `camelot.admin.action.list_action.PrintPreview`
- `camelot.admin.action.list_action.SelectAll`
- `camelot.admin.action.list_action.ImportFromFile`
- `camelot.admin.action.list_action.ReplaceFieldContents`

Inspiration

- Implementing actions as generators was made possible with the language functions of **PEP 342**.
- The EuroPython talk of Erik Groeneveld inspired the use of these features. (http://ep2011.europython.eu/conference/talks/beyond-python-enhanched-generators)
- Action steps were introduced to be able to take advantage of the new language features of **PEP 380** in Python 3.3

Documents and Reports

Generate documents Generating reports and documents is an important part of any application. Python and Qt provide various ways to generate documents. Each of them with its own advantages and disadvantages.

Method	Advantages	Disadvantages
PDF documents through reportlab	• Perfect control over layout • Excellent for mass creation of documents	• Relatively steep learning curve • User cannot edit document
HTML	• Easy to get started • Print preview within Camelot • No dependencies	• Not much layout control • User cannot edit document
Docx Word documents	• User can edit document	• Proprietary format • Word processor needed

Camelot leaves all options open to the developer.

Please have a look at *Creating a Report with Camelot* to get started with generating documents.

Generating a document or report is nothing more than yielding the appropriate action step during the `model_run()` method of an `Action`.

Action steps usable for reporting are :

- `camelot.view.action_steps.print_preview.PrintPreview`
- `camelot.view.action_steps.print_preview.PrintHtml`
- `camelot.view.action_steps.print_preview.PrintJinjaTemplate`
- `camelot.view.action_steps.open_file.OpenFile`
- `camelot.view.action_steps.open_file.OpenStream`
- `camelot.view.action_steps.open_file.OpenJinjaTemplate`

HTML based documents

```
class MovieSummary( Action ):

    verbose_name = _('Summary')

    def model_run(self, model_context):
        from camelot.view.action_steps import PrintHtml
        movie = model_context.get_object()
        yield PrintHtml( "<h1>This will become the movie report of %s!</h1>" % movie.title )
```

The supported html subset is documented here :

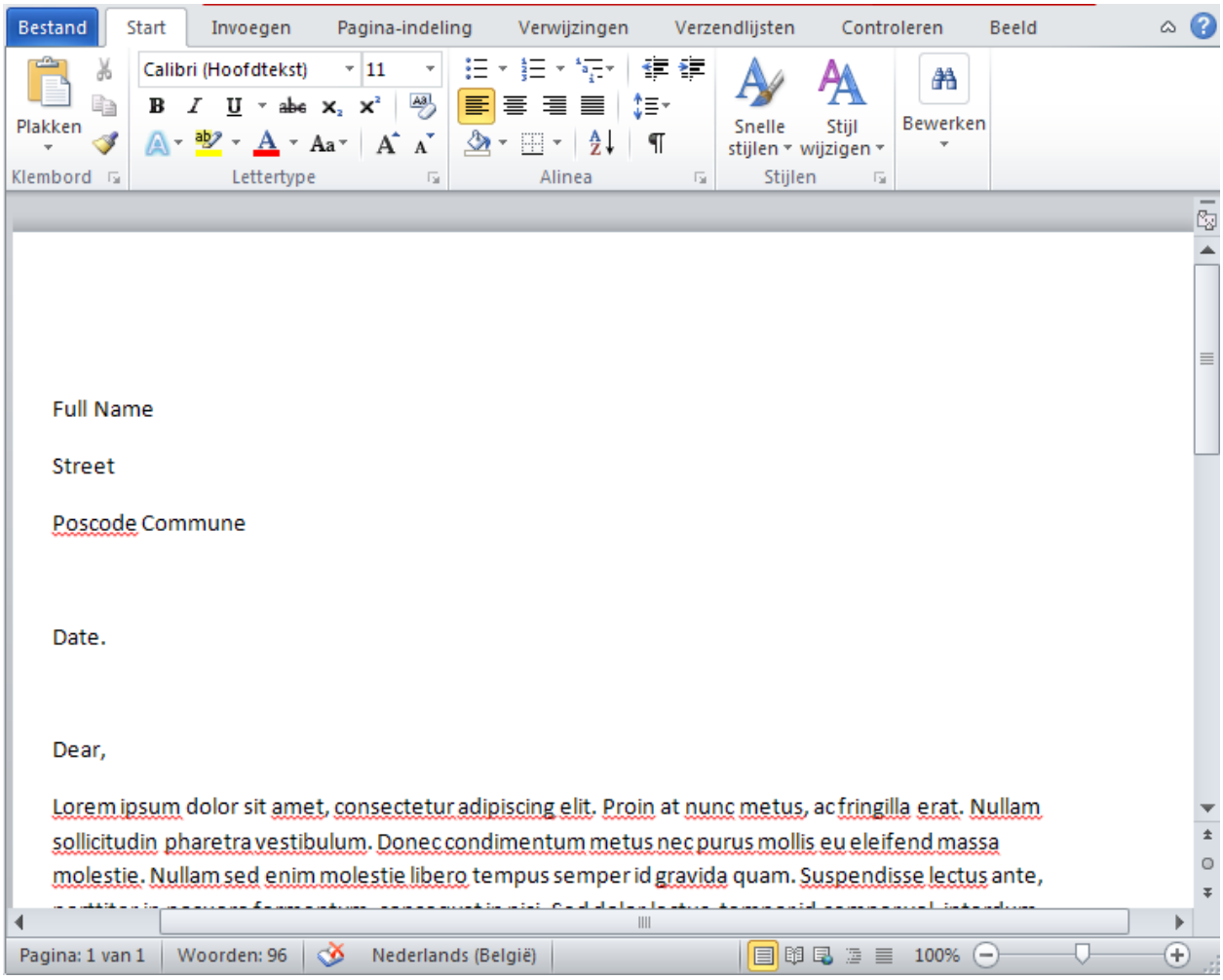
http://doc.qt.nokia.com/stable/richtext-html-subset.html

Alternative rendering Instead of `QtGui.QTextDocument` another html renderer such as `QtWebKit.QWebView` can be used in combination with the `camelot.view.action_steps.print_preview.PrintPreview` action step. The `QWebView` class has complete support for html and css.

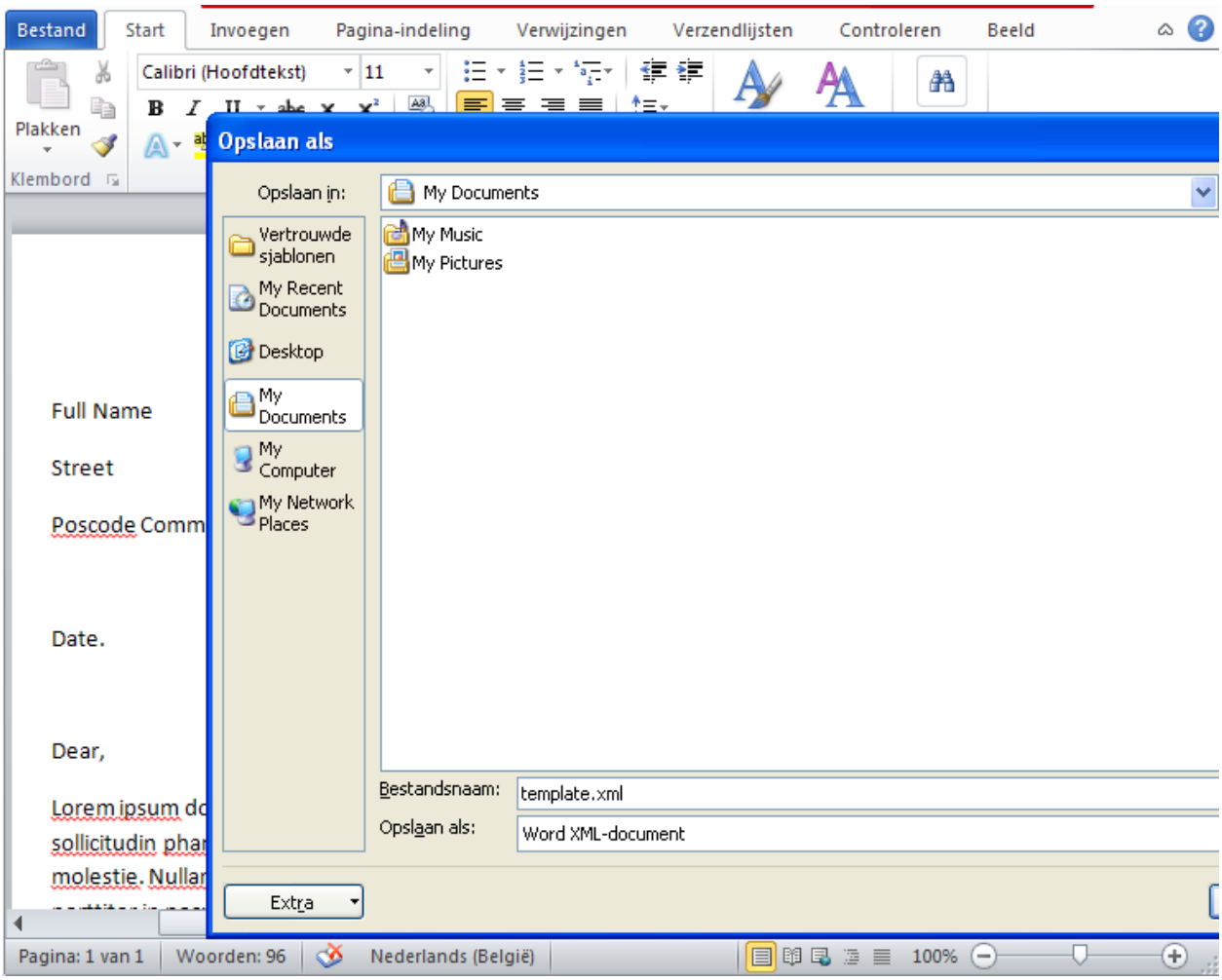
```
class WebkitPrint( Action ):

    def model_run( self, model_context ):
        from PyQt4.QtWebKit import QWebView
        from camelot.view.action_steps import PrintPreview

        movie = model_context.get_object()

        document = QWebView()
        document.setHtml( '<h2>%s</h2>' % movie.title )

        yield PrintPreview( document )
```

Docx based documents

Create a template document with MS Office Create a document using MS Office and with some placeholder text on places where you want to insert data.



And save it as an xml file :



Clean the XML generated by MS Office The XML file generated by MS Office can be cleaned using **xmllint**:

```
xmllint --format template.xml > template_clean.xml
```

Replace the placeholders The template will be merged with the objects in the selection using jinja, where the object in the selection will be available as a variable named **obj** and the time of merging the document is available as **now**:

Delegates

Delegates are a cornerstone of the Qt model/delegate/view framework. A delegate is used to display and edit data from a *model*.

In the Camelot framework, every field of an *Entity* has an associated delegate that specifies how the field will be displayed and edited. When a new form or table is constructed, the delegates of all fields on the form or table will construct *editors* for their fields and fill them with data from the model. When the data has been edited in the form, the delegates will take care of updating the model with the new data.

All Camelot delegates are subclasses of `QtGui.QAbstractItemDelegate`.

The Qt website provides detailed information the differenct classes involved in the model/delegate/view framework.

Specifying delegates The use of a specific delegate can be forced by using the `delegate` field attribute. Suppose `rating` is a field of type `integer`, then it can be forced to be visualized as stars:

```
from camelot.view.controls import delegates

class Movie( Entity ):
    title = Column( Unicode(50) )
    rating = Column( Integer )

    class Admin( EntityAdmin ):
        list_display = ['title', 'rating']
        field_attributes = {'rating':{'delegate':delegates.StarDelegate}}
```

The above code will result in:

If no *delegate* field attribute is given, a default one will be taken depending on the sqlalchemy field type.

All available delegates can be found in `camelot.view.controls.delegates`

Charts

To enable charts, **Camelot** is closely integrate with Matplotlib, one of the very high quality Python charting packages.

Often creating a chart involves gathering a lot of data, this needs to happen inside the model, to free the GUI from such tasks. Once the data is gathered, it is put into a container, this container is then shipped to the gui thread, where the chart is put on the screen.

A simple plot As shown in the example below, creating a simple plot involves two things :

1. Create a property that returns one of the chart containers, in this case the **PlotContainer** is used.
2. Specify the delegate to be used to visualize the property, this should be the **ChartDelegate**

```
from camelot.admin.object_admin import ObjectAdmin
from camelot.view.controls import delegates
from camelot.container.chartcontainer import PlotContainer

class Wave(object):

    def __init__(self):
        self.amplitude = 1
        self.phase = 0

    @property
```

```
    def chart(self):
        import math
        x_data = [x/100.0 for x in range(1, 700, 1)]
        y_data = [self.amplitude * math.sin(x - self.phase) for x in x_data]
        return PlotContainer( x_data, y_data )

    class Admin(ObjectAdmin):
        form_display = ['amplitude', 'phase', 'chart']
        field_attributes = dict(amplitude = dict(delegate=delegates.FloatDelegate,
                                                 editable=True),
                                phase = dict(delegate=delegates.FloatDelegate,
                                             editable=True),
                                chart = dict(delegate=delegates.ChartDelegate) )
```

The **PlotContainer** object takes as its arguments, the same arguments that can be passed to the matplotlib plot command. The container stores all those arguments, and later passes them to the plot command executed within the gui thread.

The simpel chart containers map to their respective matplotlib command. They include :

Actions The *PlotContainer* and *BarContainer* can be used to print or display charts as part of an action through the use of the appropriate action steps :

- `camelot.view.action_steps.print_preview.PrintChart`
- `camelot.view.action_steps.gui.ShowChart`

```
    class ChartPrint( Action ):

        def model_run( self, model_context ):
            from camelot.container.chartcontainer import BarContainer
            from camelot.view.action_steps import PrintChart
            chart = BarContainer( [1, 2, 3, 4],
                                  [5, 1, 7, 2] )
            print_chart_step = PrintChart( chart )
            print_chart_step.page_orientation = QtGui.QPrinter.Landscape
            yield print_chart_step
```

Advanced Plots For more advanced plots, the `camelot.container.chartcontainer.AxesContainer` class can be used. The *AxesContainer* class can be used as if it were a matplotlib *Axes* object. But when a method on the *AxesContainer* is called it will record the method call instead of creating a plot. These method calls will then be replayed by the gui to create the actual plot.

```
from camelot.admin.object_admin import ObjectAdmin
from camelot.view.controls import delegates
from camelot.container.chartcontainer import AxesContainer

class Wave(object):

    def __init__(self):
```

```
        self.amplitude = 1
        self.phase = 2.89

    @property
    def chart(self):
        import math
        axes = AxesContainer()
        x_data = [x/100.0 for x in range(1, 700, 1)]
        y_data = [self.amplitude * math.sin(x - self.phase) for x in x_data]
        axes.plot( x_data, y_data )
        axes.grid( True )
        axes.axvspan(self.phase-0.05, self.phase+0.05, facecolor='b', alpha=0.5)
        return axes

    class Admin(ObjectAdmin):
        form_display = ['amplitude', 'phase', 'chart']
        field_attributes = dict(amplitude = dict(delegate=delegates.FloatDelegate,
                                                 editable=True),
                                phase = dict(delegate=delegates.FloatDelegate,
                                             editable=True),
                                chart = dict(delegate=delegates.ChartDelegate) )
```

More For more information on the various types of plots that can be created, have a look at the Matplotlib Gallery.

When the AxesContainer does not provide enough flexibility, for example when the plot needs to manipulated through its object structure, more customization is possible by sub-classing either the `camelot.container.chartcontainer.AxesContainer` or the `camelot.container.chartcontainer.FigureContainer`:

Document Management

Camelot provides some features for the management of documents. Notice that documents managed by Camelot are stored in a specific location (either an application directory on the local disk, a network share or a remote server).

This in contrast with some application that just store the link to a file in the database, and don't store the file itself.

Three concepts are important for understanding how Camelot handles documents :

- The **Storage** : this is the place where Camelot stores its documents, by default this is a directory on the local system. When a file is checked in into a storage, a StoredFile is returned. Files are checked out from the storage by their StoredFile representation.
- The **StoredFile** : a stored file is a representation of a file stored in a storage. It does not contain the file itself but its name and meta information.
- The **File** Field type : is a custom field type to write and read the StoredFile into the database. The actual name of the StoredFile is the only thing stored in the database.

The File field type Usually the first step when working with documents is to use the File field type somewhere in the model definition. Alternatively the Image field type can be used if one only wants to store images in that field.

The StoredFile When the File field type is used in the code, it returns and accepts objects of type StoredFile.

The Image field type will return objects of type StoredImage.

The Storage This is where the actual file is stored. The default storage implementation simply represents a directory on the file system.

Under the hood

A lot of things happen when a Camelot application starts up. In this section we give a brief overview of those which might need to be adapted for more complex applications

Global settings Camelot has a global *settings* object of which the attributes are used throughout Camelot whenever a piece of global configuration is needed. Examples of such global configuration are the location of the database and the location of stored files and images. To access the global configuration, simply import the object

```
from camelot.core.conf import settings
print settings.CAMELOT_MEDIA_ROOT()
```

To manipulate the global configuration, create a class with the needed attributes and methods and append it to the global configuration :

The *settings* object should have a method named `ENGINE`, uses the create_engine SQLAlchemy function to create a connection to the database. Camelot provides a default `sqlite` URI scheme. But you can set your own.

```
    def ENGINE( self ):
        from sqlalchemy import create_engine
        return create_engine(u'sqlite:///%s/%s'%( self.data_folder,
                                                  self.data ) )
```

Older versions of Camelot looked for a *settings* module on *sys.path* to look for the global configuration. This approach is still supported.

Setting up the ORM When the application starts up, the *setup_model* method of the *Settings* class is called. In this function, all model files should be imported, to make sure the model has been completely setup. The importing of these files is enough to define the mapping between objects and tables.

The import of these model definitions should happen before the call to *create_all* to make sure all models are known before the tables are created.

Setting up the Database

Engine The *Settings* class should contain a method named *ENGINE* that returns a connection to the database. Whenever a connection to the database is needed, this method will be called. The `camelot.core.conf.SimpleSettings` has a default *ENGINE* method that returns an SQLite database in a user directory.

Metadata *SQLAlchemy* defines the `MetaData` class. A *MetaData* object contains all the information about a database schema, such as Tables, Columns, Foreign keys, etc. The `camelot.core.sql` contains the singleton *metadata* object which is the default `MetaData` object used by Camelot. In the *setup_model* function, this *metadata* object is bound to the database engine.

In case an application works with multiple database schemas in parallel, this step needs to be adapted.

Creating the tables By simply importing the modules which contain parts of the model definition, the needed table information is added to the *metadata* object. At the end of the *setup_model* function, the *create_all* method is called on the metadata, which will create the tables in the database if they don't exist yet.

Working without the default model Camelot comes with a default model for Persons, Organizations, History tracking, etc.

To turn these on or off, simply add or remove the import statements of those modules from the *setup_model* method in the *Settings* class.

Transactions Transactions in Camelot can be used just as in normal SQLAlchemy. This means that inside a `camelot.admin.action.base.Action.model_run()` method a transaction can be started and committed

```
with model_context.session.begin()
    ...do some modifications...
```

More information on the transactional behavior of the session can be found in the SQLAlchemy documentation ...

Using Camelot without the GUI Often a Camelot application also has a non GUI part, like batch scripts, server side scripts, etc.

It is of course perfectly possible to reuse the whole model definition in those non GUI parts. The easiest way to do so is to leave the Camelot GUI application as it is and then in the non GUI script, initialize the model first

```
from camelot.core.conf import settings
settings.setup_model()
```

From that point, all model manipulations can be done. Access to the single session can be obtained from anywhere through the *Session* factory method

```
from camelot.core.orm import Session
session = Session()
```

After the manipulations to the model have been done, they can be flushed to the db

```
session.flush()
```

Built in data models

Camelot comes with a number of built in data models. To avoid boiler plate models needed in almost any application (like Persons, Addresses, etc.), the developer is encouraged to use these data models as a start for developing custom applications.

Modules The `camelot.model` module contains a number of submodules, each with a specific purpose

To activate such a submodule, the submodule should be imported in the *setup_model* method of *settings* class, before the tables are created

```
def setup_model( self ):
    from camelot.core.sql import metadata
    metadata.bind = self.ENGINE()
    from camelot.model import authentication
    from camelot.model import party
    from camelot.model import i18n
    from camelot.core.orm import setup_all
    setup_all( create_tables=True )
```

Persons and Organizations

I18N

Fixture

Authentication

Batch Jobs A batch job object can be used as a context manager :

```
from camelot.model.batch_job import BatchJob, BatchJobType
synchronize = BatchJobType.get_or_create( u'Synchronize' )
with BatchJob.create( synchronize ) as batch_job:
    batch_job.add_strings_to_message( [ u'Synchronize part A',
                                        u'Synchronize part B' ] )
    batch_job.add_strings_to_message( [ u'Done' ], color = 'green' )
```

Whenever an exception happens inside the *with* block, the stack trace of this exception will be written to the bach job object and it's status will be set to *errors*. At the end of the *with* block, the status of the batch job will be set to *finished*.

History tracking

Customization

Adding fields Sometimes the built in models don't have all the fields or relations required for a specific application. Fortunately it is possible to add fields to an existing model on a per application base.

To do so, simply assign the required fields in the application specific model definition, before the tables are created.

```
party.Person.language = schema.Column( types.Unicode(30) )

metadata.create_all()
p = party.Person( first_name = u'Peter',
                  last_name = u'Principle',
                  language = u'English' )
session.flush()
```

Fixtures : handling static data in the database

Some tables need to be filled with default data when users start to work with the application. The Camelot fixture module `camelot.model.fixture` assist in handling this kind of data.

Suppose we have an entity *PartyCategory* to divide Persons and Organizations into certain groups.

The complete definition of such an entity can be found in `camelot.model.authentication.PartyCategory`.

To make things easier for the first time user, some prefab categories should be available when the user starts the application. Such as *Suspect*, *Prospect*, *VIP*.

When to update fixtures Most of the time static data should be created or updated right after the model has been set up and before the user starts using the application.

The easiest place to do this is in the setup_model method inside the settings.py module.

So we rewrite settings.py to include a call to a new update_fixtures method:

```
def update_fixtures():
  """Update static data in the database"""
  from camelot.model.fixture import Fixture
  from model import MovieType

def setup_model():
  from camelot.model import *
  from camelot.model.memento import *
  from camelot.model.synchronization import *
  from camelot.model.authentication import *
  from camelot.model.i18n import *
  from camelot.model.fixture import *
  from model import *
  setup_all(create_tables=True)
  updateLastLogin()
  update_fixtures()
```

Creating new data When creating new data with the fixture module, a reference to the created data will be stored in the fixture table along with a 'fixture key'. This fixture key can be used later to retrieve or update the created data.

So lets create some new movie types:

```
def update_fixtures():
  """Update static data in the database"""
  from camelot.model.fixture import Fixture
  from model import MovieType
  Fixture.insertOrUpdateFixture(MovieType,
                                fixture_key = 'comic',
                                values = dict(name='Comic'))
  Fixture.insertOrUpdateFixture(MovieType,
                                fixture_key = 'scifi',
                                values = dict(name='Science Fiction'))
```

Fixture keys should be unique for each Entity class.

Update fixtures When a new version of the application gets released, we might want to change the static data and add some icons to the movie types. Thanks to the 'fixture key', it's easy to retrieve and update the already inserted data, just modify the update_fixtures function:

```
def update_fixtures():
  """Update static data in the database"""
  from camelot.model.fixture import Fixture
  from model import MovieType
  Fixture.insertOrUpdateFixture(MovieType,
                                fixture_key = 'comic',
                                values = dict(name='Comic', icon='spiderman.png'))
  Fixture.insertOrUpdateFixture(MovieType,
                                fixture_key = 'scifi',
                                values = dict(name='Science Fiction', icon='light_saber.png'))
```

The fixture version In case lots of data needs to be read into the database (like a list of postal codeds), it might make no sense to create a new fixture for each code, instead a fixture version number can be set to indicate a list has been read into the database. The `camelot.model.fixture.FixtureVersion` exists to facilitate this.

```
import csv
if FixtureVersion.get_current_version( u'demo_data' ) == 0:
    reader = csv.reader( open( example_file ) )
    for line in reader:
        Person( first_name = line[0], last_name = line[1] )
    FixtureVersion.set_current_version( u'demo_data', 1 )
    session.flush()
```

Managing a Camelot project

Once a project has been created and set up as described in the tutorial *Creating a Movie Database Application*, it needs to be maintained and managed over time.

The command line tool **camelot_admin.py** exist to assist in the management of Camelot projects.

camelot_admin.py

The Two Threads

Most users of Camelot won't need the information in this Chapter and can simply enjoy building applications that don't freeze. However, if you start customizing your application beyond developing custom delegates, this information might be crucial to you.

Introduction A very important aspect of any GUI application is the speed with which it responds to the user's request. While it is acceptable that some actions take some time to complete, an application freezing for even half a second makes the user feel uncomfortable.

From an application developer's point of view, potential freezes are everywhere (open a file, access a database, do some calculations), so we need a structural approach to get rid of them.

Two different approaches are possible. The first approach is split all possibly blocking operations into small parts and hook everything together with events. This is the approach taken in some of the QT classes (eg.: the network classes) or in the Twisted framework. The second approach is to use multiple threads of execution and make sure the blocking operations run in another thread than the GUI.

Events :

- No multi-threaded programming needed : no deadlocks etc.
- Every single library you use must support this approach

Multiple threads :

- Scary : potential race conditions and deadlocks
- Can be used with existing libraries

The Camelot framework was developed using the multi-threaded approach. This allows to build on top of a large number of existing libraries (sqlalchemy, PIL, numpy,...) that don't support the event based approach.

Two Threads To keep the problems associated with multi-threaded programming under control, Camelot runs only two threads for its basic operations. Those threads don't share any data with each other and exchange information using a message queue (the way Erlang advocates). This ensures there are no deadlocks or race conditions.

The first thread, called the GUI Thread contains the QT widgets and runs the QT event loop. No blocking operations should take place in this thread. The second thread contains all the data, like objects mapped to the database by sqlalchemy, and is called the Model Thread.

This approach keeps the problem of application freezes under control, it won't speed up your application when certain actions take a long time, but it will ensure the gui remains responsive during those actions.

The Model Thread Since every single operation on a data model is potentially blocking (eg : getting an attribute of a class mapped to the database by sqlalchemy might trigger a query to the database which might be overloaded at that time), the whole data model lives in a separate thread and every operation on the data model should take place within this thread.

To keep things simple and avoid the use of locks and data synchronization between threads, there is only one such thread, called the Model Thread.

Other threads that want to interact with the model can post operations to the model thread using its queue

```
from camelot.view.model_thread import get_model_thread

mt = get_model_thread()
mt.post(my_operation)
```

where 'my_operation' is a function that will then be executed within the model thread.

The GUI Thread Now that all potentially blocking operations have been move to the model thread, we have a GUI Thread that never blocks. But the GUI thread will need some data from the model to present to the user.

The GUI thread gets its data by posting an operation to the Model Thread that strips some data from the model, this data will then be posted by the Model thread to the GUI thread.

Suppose we want to display the name of the first person in the database in a QLabel

```
from camelot.view.model_thread import get_model_thread
from PyQt4 import QtGui

class PersonLabel(QtGui.QLabel):

  def __init__(self):
QtGui.QLabel.__init__(self)
mt = get_model_thread()
mt.post(self.strip_data_from_model, self.put_data_on_label)
```

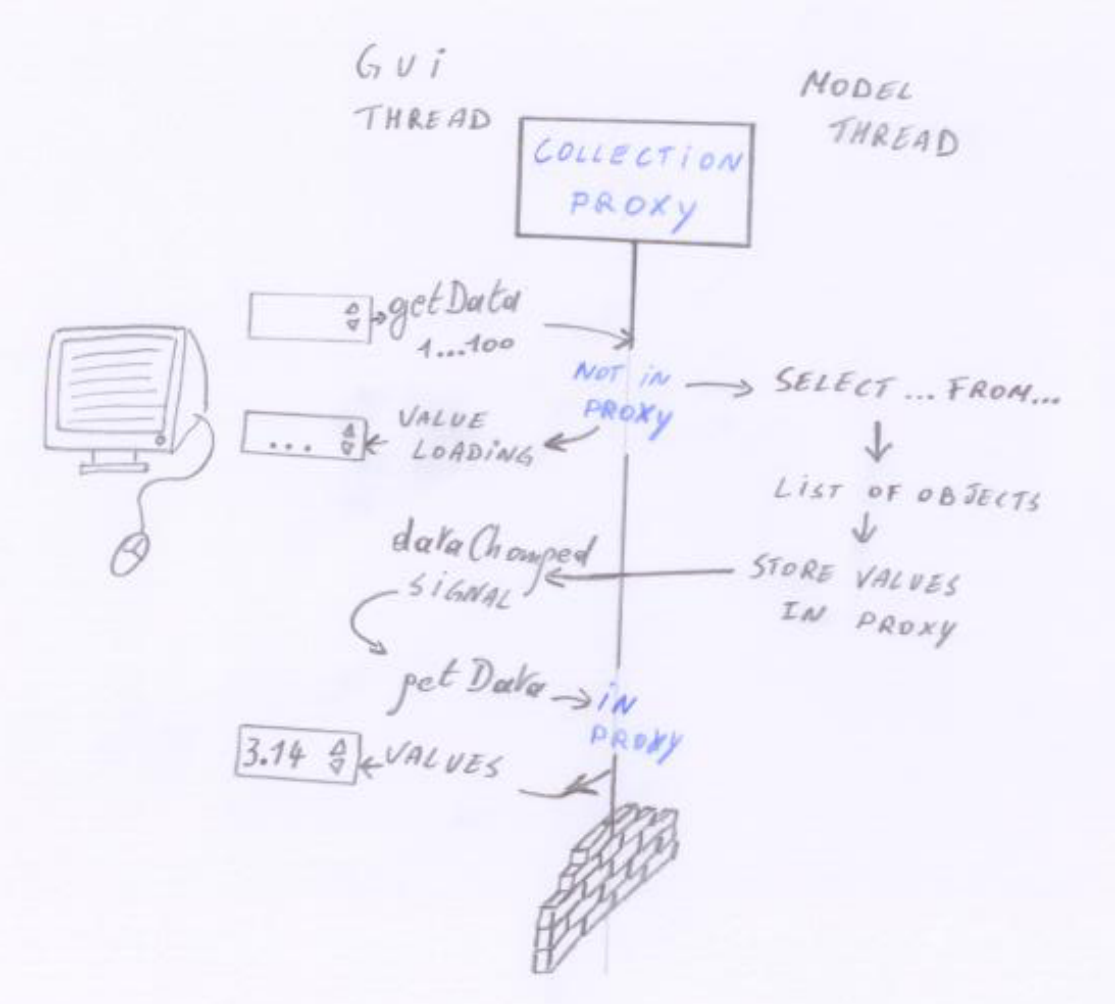
```
  def strip_data_from_model(self):
from camelot.model.authentication import Person
return Person.query.first().name

  def put_data_on_label(self, name):
    self.setText(name)
```

When the strip_data_from_model method is posted to the Model Thread, it will be executed within the Model Thread and its result (the name of the person) will be posted back to the GUI thread. Upon arrival of the name in the GUI thread the function put_data_on_label will be executed within the GUI thread with as its first argument the name.

In reality, the stripping of data from the model and presenting this data to the gui is taken care off by the proxy classes in camelot.view.proxy.

Actions

Proxy classes



Application speedup

Frequently Asked Questions

How to use the PySide bindings instead of PyQt ? The Camelot sources as well as the example videostore application can be converted from PyQt applications to PySide with the *camelot_admin* tool.

Download the sources and position the shell in the main directory, and then issue these commands:

```
python camelot/bin/camelot_admin.py to_pyside .
```

This will create a subdirectory 'to_pyside' which contains the converted source code.

Can I use Camelot with an existing database ? Both Declarative and Camelot can be used with an existing schema. However, since Camelot acts on objects, the classes for those objects still need to be defined.

Here's a short example of using camelot with an existing database :

```
from sqlalchemy.engine import create_engine
from sqlalchemy.pool import StaticPool

engine = create_engine( 'sqlite:///test.sqlite' )
#
# Create a table in the database using plain old sql
#
connection = engine.connect()
try:
    connection.execute("""drop table person""")
except:
    pass
connection.execute( """create table person ( pk INTEGER PRIMARY KEY,
                                             first_name TEXT NOT NULL,
                                             last_name TEXT NOT NULL )""" )
connection.execute( """insert into person (first_name, last_name)
                       values ("Peter", "Principle")""" )

#
# Use declarative to reflect the table and create classes
#
from camelot.admin.entity_admin import EntityAdmin
from camelot.core.sql import metadata
from sqlalchemy.schema import Table
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base( metadata = metadata )

class Person( Base ):
    __table__ = Table( 'person', Base.metadata,
                       autoload=True, autoload_with=engine )

    class Admin( EntityAdmin ):
        list_display = ['first_name', 'last_name']

#
# Setup a camelot application
#
from camelot.admin.application_admin import ApplicationAdmin
from camelot.admin.section import Section
from camelot.core.conf import settings
```

```
class AppAdmin( ApplicationAdmin ):

    def get_sections( self ):
        return [ Section( 'All tables', self, items = [Person] ) ]

class Settings(object):

    def ENGINE( self ):
        return engine

    def setup_model( self ):
        metadata.bind = engine

settings.append( Settings() )
app_admin = AppAdmin()

#
# Start the application
#
if __name__ == '__main__':
    from camelot.view.main import main
    main( app_admin )
```

More information on using Declarative with an existing database schema can be found in the Declarative documentation.

Why is there no *Save* button ? Early on in the development process, the controversial decision was made not to have a *Save* button in Camelot. Why was that ?

- User friendlyness. One of the major objectives of Camelot is to be user friendly. This also means we should reduce the number of 'clicks' a user has to do before achieving something. We believe the 'Save' click is an unneeded click. The application knows when the state of a form is valid for persisting it to the database, and can do so without user involvement. We also want to take the 'saving' issue out of the mind of the user, he should not bother wether his work is 'saved', it simply is.
- Technical. Once you decide to use a *Save* button, you need to ask yourself where you will put that button and what its effect will be. This question becomes difficult when you want to enable the user to edit a complex datastructure with one-to-many and many-to-many relations. Most applications solve this by limiting the options for the user. For example, most accounting packages will not allow you to create a new customer when you are creating a new invoice. Because when you save the invoice, should the customer be saved as well ? Or should the customer have it's own save button ? Those packages therefor require the user to first create a customer, and only then can an invoice be created. These are limitation we don't want to impose with Camelot.
- Consistency between editing in table or form view. We wanted the table view to be really easy to edit (to behave a bit like a spreadsheet), so it's easy for the user to do bulk updates. As such the user should not be bothered by pressing the *Save* button all the time. If there is no need to save in the table view, there should be no need in the form view either.

Some couter arguments for this decision are :

- But what if the user wants to 'modify' a form and not save those changes ? This is indeed something that is not possible without a *Save* and it accompanying *Cancel* button. But this is something a developer will do a lot while testing an application, but is outside of the normal workflow of a user. Most users typically want to enter or modify as much data as possible, they are not testing the application to see how it would behave on certain data input.
- A form should be validated before it is saved. In an application there are two levels of validation. The first level is to validate before something is persisted into the database, this can be done in Camelot using a custom

implementation of a `camelot.admin.validator.entity_validator.EntityValidator`. The second level is a validation before the entered data can be used in the business process. To do this second level validation, one can use state changes (Action buttons that change the state of a form, eg from 'Draft' to 'Complete'). A good example of this is when entering a booking into an accounting package. When a booking is entered, it can only be used when debit equals credit. What would happen when this validation is done at the moment the form is 'saved'. Suppose a user has been working for the better part of the day on a complex booking, but is not done yet at the end of the day. Since he cannot yet save his work he has two options, discard it and restart the next day, or enter some bogus data to be able to save it. What will happen in the later case when his manager is creating a report a bit later. So the correct situation in this case is having your work saved at all times, and to put your booking from a 'draft' state to a 'complete' state once its ready. This state change will then check if debit equals credit.

Two years after we made this move, Apple decided to follow our example : http://www.apple.com/macosx/whats-new/auto-save.html

But my users really want a *Save* button ? We advise you to listen very well to the arguments the user has for wanting a *Save* button. You will be able to solve most of them by using state changes instead of a *Save* button. The other arguments probably have to do with expections users have from using other applications, as for those simply ask the users to try to work for a week without a *Save* button and get back to you if after that week, they still have issues with it. Please let us know when they do !

Advanced Topics

This is documentation for advanced usage of the Camelot library.

Internationalization

The Camelot translation system is a very small wrapper around the Qt translation system. Internally, it uses the `QCoreApplication.translate()` method to do the actual translation.

On top of that, it adds the possibility for end users to change translations theirselves. Those translations are stored in the database. This mechanism can be used to adapt the vocabulary of an application to that of a specific company.

How to Specify Translation Strings Translation strings specify "This text should be translated.". It's your responsibility to mark translatable strings; the system can only translate strings it knows about.

```
from camelot.core.utils import ugettext as _

message = _("Hello brave new world")
```

The above example translates the given string immediately. This is not always desired, since the message catalog might not yet be loaded at the time of execution. Therefore translation strings can be specified as lazy. They will only get translated when they are used in the GUI.

```
from camelot.core.utils import ugettext_lazy as _

message = _("This translation is delayed")
```

Translation strings in model definitions should always be specified as lazy translation strings. Only lazy translation strings can be translated by the end user in various forms.

Translating Camelot itself To extract translation files from the Camelot source code, Babel needs to be installed.

In the root folder of the Camelot source directory.

First update the translation template:

```
python setup.py extract_messages
```

If your language directory does not yet exists in 'camelot/art/translations':

```
python setup.py init_catalog --locale nl
```

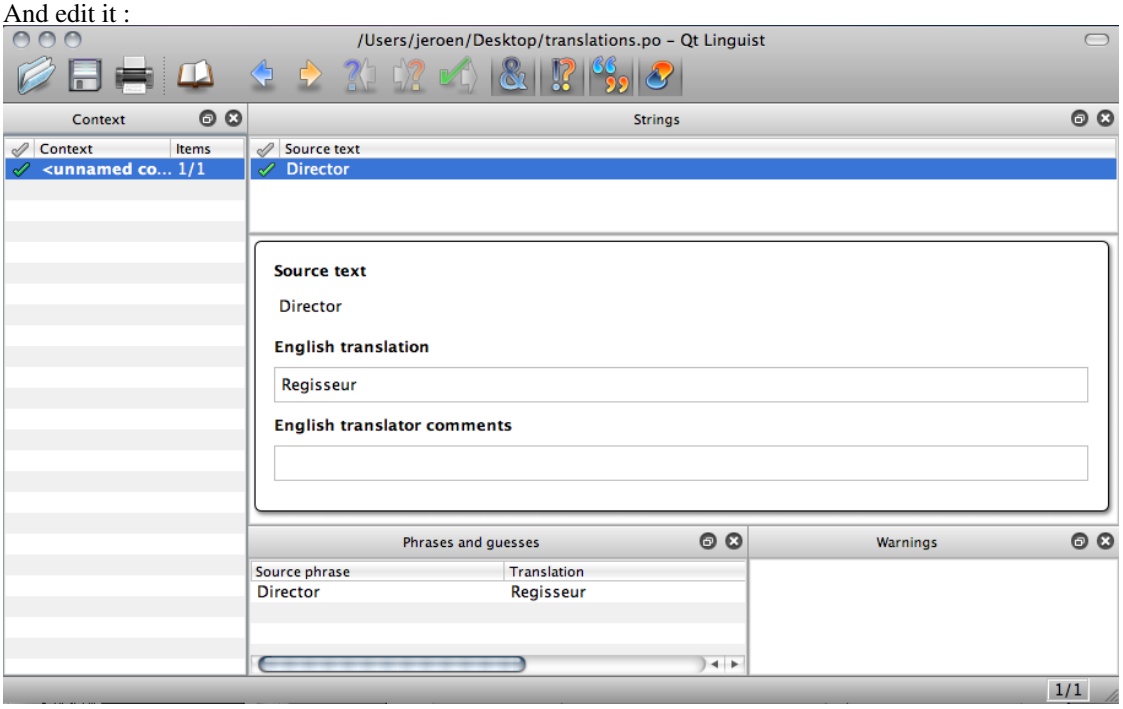
If it allready exists, update it from the translation template:

```
python setup.py update_catalog
```

In the language subdirectory of 'camelot/art/translations', there is a subdirectory 'LC_MESSAGES' which contains the .po translation file. This translation file can then be translated with linguist

```
linguist camelot.po
```

And edit it :



Make sure to save them back as GNU gettext .po files.

Then the .po file should be converted to a .qm file to make it loadable at run time:

```
lrelease camelot.po
```

Don't forget to post your new .po file on the mailing list, so it can be included in the next release.

For more background information, please have a look at the Babel Documentation

Where to put Translations Translations can be put in 2 places :

- in po files which have to be loaded at application startup

- in the Translation table : this table is editable by the users via the Configuration menu. This is the place to put translations that should be editable by the users. At application startup, all records in this table related to the current language will be put in memory.

Loading translations Translations are loaded when the application starts. To enforce the loading of the correct translation file, one should overwrite the `camelot.admin.application_admin.ApplicationAdmin.get_translator()` method. This method should return the proper `QtCore.QTranslator` object.

End user translations Often it is convenient to let the end user create or update the translations of an application, this allows the end user to put a lot of domain knowledge into the application.

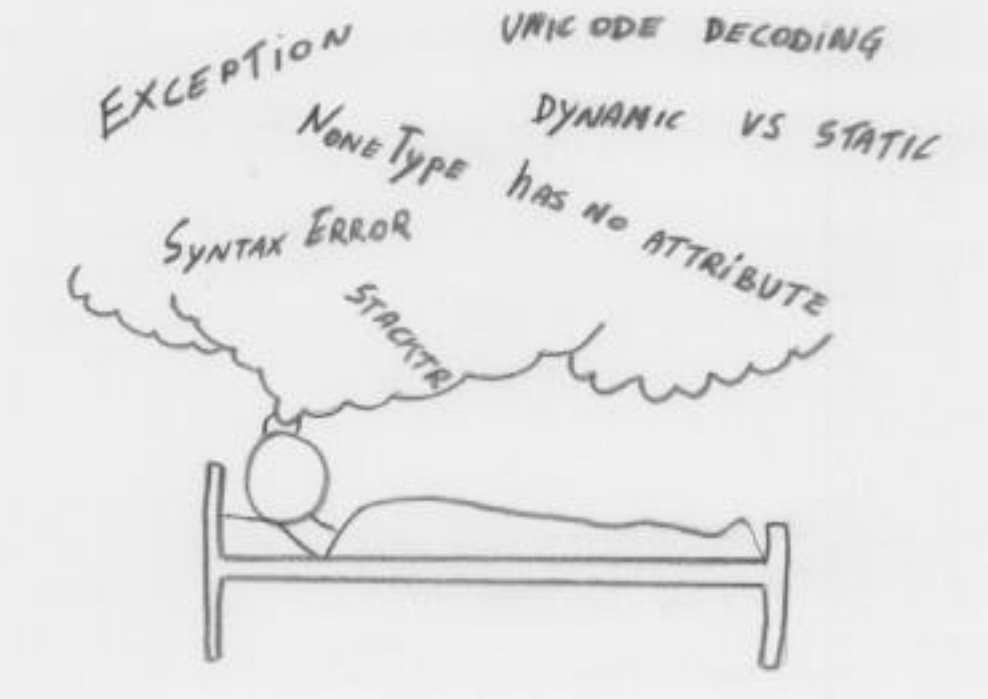
Therefore, all lazy translation strings can be translated by the end user. When the user right-clicks on a label in a form, he can select *Change translation* from the menu and update the current translation (for the current language). This effectively updates the content of the **Translation** table.

After some time the developer can take a copy of this table and decide to put these translations in po files.

Unittests

Release default

Date April 23, 2013



Deployment

After developing a Camelot application comes the need to deploy the application, either at a central location or in a distributed setup.

Building .egg files Whatever the deployment setup is, it is almost always a good idea to distribute your application as a single .egg file, containing as much as possible the dependencies that are likely to change often during the lifetime of the application. Resource files (like icons or templates can be included in this .egg file as well).

Building .egg files is a relatively straightforward process using setuptools.

When a new Camelot project was created with *camelot_admin*, a `setup.py` file was made that is able to build eggs using this command

```
python -O setup.py bdist_egg --exclude-source-files
```

Note: The advantage of using .egg files comes when updating the application, simply replacing a single .egg file at a central location is enough to migrate all your users to the new version.

Windows deployment

Through CloudLaunch CloudLaunch is a service to ease the deployment and update process of Python applications. It's main features are :

- Building Windows Installers
- Updating deployed applications
- Monitoring of deployed applications

As CloudLaunch is build on top of setuptools, it works with .egg files, CloudLaunch works cross platform, so it's perfectly possible to build a Windows installer, or update a Windows application from Linux.

To build a .egg file that can be deployed through CloudLaunch, use the command:

```
python.exe setup.py bdist_cloud
```

This will create 2 files in the dist/cloud folder, a traditional .egg file and a .cld file. The .egg file is a normal .egg file with some additional metadata included, and without sources. The .cld file contains metadata of the .egg file, such as its checksum, and information on how get updated versions of the .egg once deployed.

To make sure the application will run smoothly once deployed, one should test if the generated .egg and .cld combination works:

```
cd dist\cloud
cloudlaunch.exe --cld-file movie_store.cld
cd ..\..
```

If this is working, a Windows installer can be build:

```
python.exe setup.py bdist_cloud wininst_cloud
```

This will generate a movie_store.exe file in distcloud, which is an installer for your application. The end user can now install and run your application on his machine.

Now is the time to monitor the application as it runs on the end user machine:

```
python.exe setup.py monitor_cloud
```

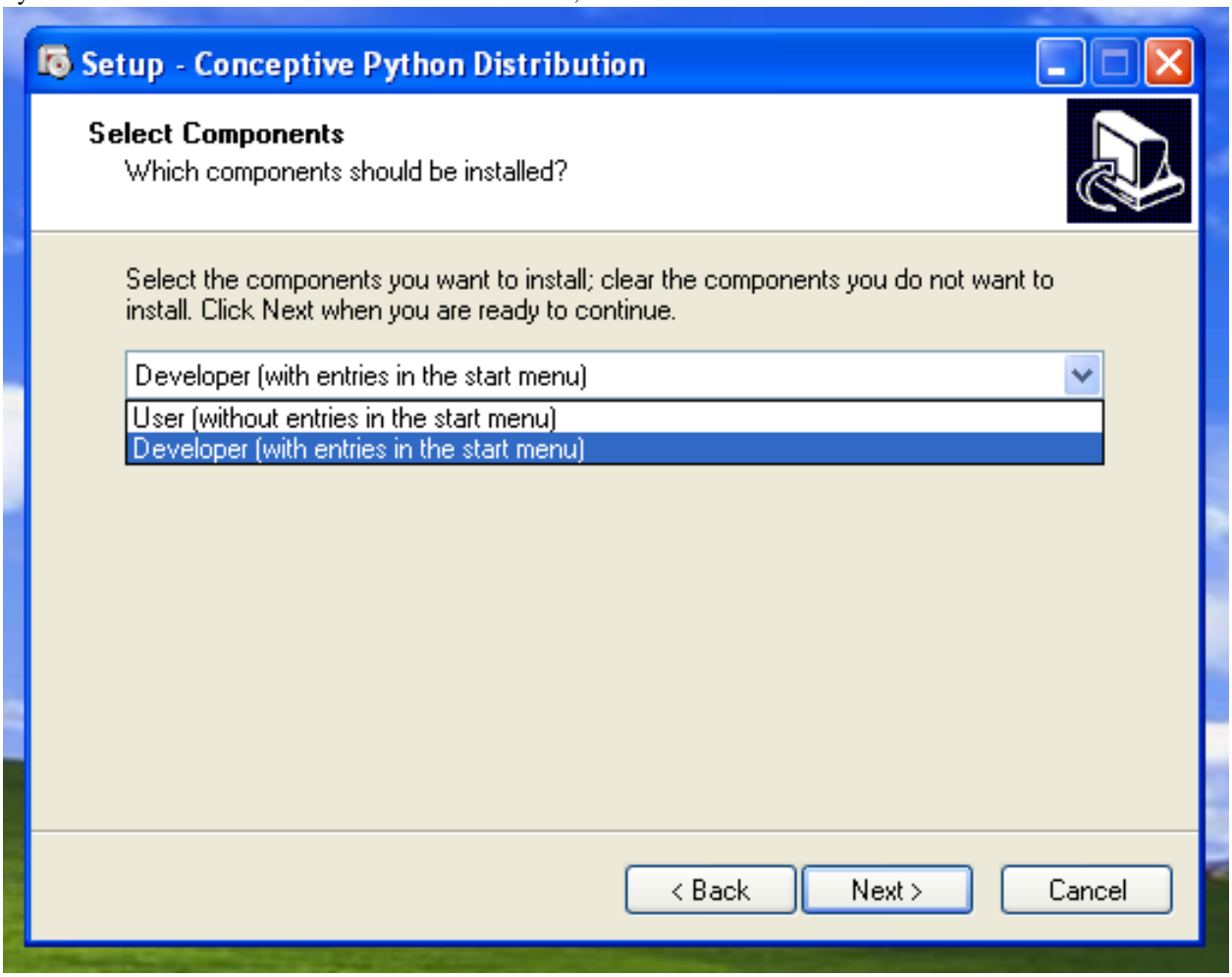
Will display all the logs issued on the end user machine if that machine is connected to the internet.

When development of the application continues, it will be needed to present the user with an updated version of the application. This is done with the command:

```
python.exe setup.py bdist_cloud upload_cloud
```

This will send an updated .egg and .cld file to the central repository, where the end-user application will check for updates. If such an update is detected, the application will download the new egg and run from that one.

Using .egg files First of all python needs to be available on the machines that are going to run the application. The easies way to achieve this is by installing the Conceptive Python Distribution (CPD) on the target machine. This Python distribution can be installed in **End user mode**, which means the user will not notice it is installed.



Notice that for python to be available, it not necessarily needs to be installed on every machine that runs the application. Installing python on a shared disk of a central server might just be enough.

Also put the .egg file on a shared drive.

Then, the easiest way to proceed is to put a little .vbs bootstrap script on the shared drive and put shortcuts to it on the desktops of the users. The .vbs script can look like this:

```
Set WshShell = WScript.CreateObject("WScript.Shell")
WshShell.Environment("Process").item("PYTHONPATH") = "R:\movie_store-01.01-py2.7.egg;"
WshShell.Run """C:\Program Files\CPD\pythonw.exe"" -m movie_store.main"
```

Linux deployment The application can be launched by putting the .egg in the PYTHONPATH and starting python with the -m option:

```
export PYTHONPATH = /mnt/r/movie_store-01.01-py2.7.egg
python.exe -m movie_store.main
```

Don't forget that all dependencies for your application should be installed on the system or put in the PYTHONPATH

Authentication and permissions

fine grained authentication and authorization is not yet included as part of the Camelot framework.

what is included is the function :

`camelot.model.authentication.get_current_authentication()`

which returns an object of type :class:'camelot.model.authentication.AuthenticationMechanism

where the username is the username of the currently logged in user (because on most desktop apps, you don't want a separate login process for your app, but rely on that of the OS).

this function can then be used if you build the *Admin* classes for your application :

- set the *editable* field attribute to a function that only returns Thrue when the current authentication requires editing of fields
- in the *ApplicationAdmin.get_sections method*, to hide/show sections depending on the logged in user
- in the *EntityAdmin* subclasses, in the *get_field_attributes* method, to set fields to editable=False/True depending on the logged in user

Development Guidlines

Date April 23, 2013

Python, PyQt and Qt objects Python and Qt both have their own way of tracking objects and deleting them when they are no longer needed :

- Python does reference counting supported by a garbage collector.
- Qt has parent child relations between objects. When a parent object is deleted, all its child objects are deleted as well.

PyQt merges these two concepts by introducing **ownership** of objects :

- Pure python objects are owned by Python, Python takes care of their deletion.
- Qt objects wrapped by Python are either:
  - owned by Qt when they have a parent object, Qt will delete them, when their parent object is deleted
  - owned by Python when they have no parent, Python will delete them, and trigger the deletion of all their children by Qt
- Qt objects that are not wrapped by Python, those are in one way or another children of a Qt object that is wrapped by Python, they will get deleted by Qt.

The difficult case in this scheme is the case where Qt objects are wrapped by Python but have a parent object. This can happen in two ways :

- A Qt object is created in python, but with a parent

```
from PyQt4 import QtCore

parent = QtCore.QObject()
child = QtCore.QObject(parent=parent)
```

In this case PyQt is able to track when the object is deleted by Qt and raises exceptions accordingly when a method of underlying Qt object is called after the deletion

```
parent = QtCore.QObject()
child = QtCore.QObject(parent=parent)
del parent
print child.objectName()
```

will raise a RuntimeError: underlying C/C++ object has been deleted.

- A Qt object is returned from a Qt function that created the object without Python being aware of it. When the object is passed as a return value PyQt will wrap it as a Python object, but is unable to track when Qt deletes it

```
from PyQt4 import QtGui
app = QtGui.QApplication([])
window = QtGui.QMainWindow()
statusbar = window.statusBar()
del window
statusbar.objectName()
```

Will result in a segmentation fault.

A segmentation fault will happen in several cases :

- Python tries to delete a Qt object already deleted by Qt
- PyQt calls a function of a Qt object already deleted
- Qt calls a function of a Qt object already deleted by Python

In principle, PyQt is able to handle all cases where the object has been created by Python. However, when this ownership tracking is combined with threading and signal slot connections, a lot of corner cases arise in both Qt and PyQt.

To play on safe, these guidelines are used when developing Camelot :

- Never keep a reference to objects created by Qt having a parent, so only use:

```
window.statusBar().objectName()
```

- Keep references to Qt child objects as short as possible, and never beyond the scope of a method call. This is possible because qt allows objects to have a name.

  so instead of doing

```
from PyQt4 import QtGui

class Parent( QtGui.QWidget ):

        def __init__( self ):
                super(Parent, self).__init__()
                self._child = QtGui.QLabel( parent=self )

        def do_something( self ):
                print self._child.objectName()
```

this is done

```
from PyQt4 import QtGui

class Parent( QtGui.QWidget ):

        def __init__( self ):
                super(Parent, self).__init__()
                child = QtGui.QLabel( parent=self )
                child.setObjectName( 'label' )

        def do_something( self ):
            child = self.findChild( QtGui.QWidget, 'label' )
            if child != None:
                        print child.objectName()
```

should the child object have been deleted by Qt, the findChild method will return None, and a segmentation fault is prevented. An explicit check for None is needed, since even if the widget exists, it might evaluate to 0 or an empty string.

Debugging Camelot and PyQt

Log the SQL Queries Configure SQLAlchemy to log all queries:

```
logging.getLogger('sqlalchemy.engine').setLevel(logging.DEBUG)
```

Enable core dumps

Linux For older gdb versions (pre 7), copy the gdbinit file from the python Misc folder:

```
cp gdbinit ~/.gdbinit
```

use:

```
ulimit -c unlimited
```

load core file in gdb:

```
gdb /usr/bin/python -c core
```

In newer gdb versions, Python can run inside gdb:

http://bugs.python.org/issue8032

To give gdb python super powers:

```
(gdb) python
>import sys
>sys.path.append('Python-2.7.1/Tools/gdb/libpython.py')
>import libpython
>reload(libpython)
>
>end
```

https://fedoraproject.org/wiki/Features/EasierPythonDebugging

Windows

- Install *Debugging tools for Windows* from MSDN

Install 'Debug Diagnostic Tool'

http://stackoverflow.com/questions/27742/finding-the-crash-dump-files-for-a-c-app

http://blogs.msdn.com/b/tess/

Setup Qt Creator

http://doc.qt.nokia.com/qtcreator-snapshot/creator-debugger-engines.html

- Install Windows Sysinternals process utilities from MSDN

http://technet.microsoft.com/en-us/sysinternals/bb795533

Camelot, Qt, PyQt Licenses

Camelot License

> Camelot is Copyright (C) 2007-2013 Conceptive Engineering bvba. www.conceptive.be / info@conceptive.be
>
> You may use, distribute and copy Camelot under the terms of GNU General Public License version 2, which is displayed below.
>
> A commercial license can be obtained from Conceptive Engineering bvba Please mail to info@conceptive.be for inquiries.

> **GNU GENERAL PUBLIC LICENSE** Version 2, June 1991
>
> Copyright (C) 1989, 1991 Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.
>
> Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Lesser General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. **You may modify your copy or copies of the Program or any portion** of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

    a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

    b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

    c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the

terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

> 3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
>
> > a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
> >
> > b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
> >
> > c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. **You may not copy, modify, sublicense, or distribute the Program** except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. **You are not required to accept this License, since you have not** signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

   6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

   7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a

patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the

original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY

OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.> Copyright (C) <year> <name of author>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) year name of author Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’. This is free software, and you are welcome to redistribute it under certain conditions; type ‘show c’ for details.

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than ‘show w’ and ‘show c’; they could even be mouse-clicks or menu items–whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program ‘Gnomovision’ (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989 Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License.

Large parts of the source in camelot/core/orm/ are based on the Elixir library (http://elixir.ematia.de).

The Elixir library was released under the MIT license, with the following copyright notice :

This is the MIT license: http://www.opensource.org/licenses/mit-license.php

Copyright (c) 2007, 2008 Jonathan LaCour, Daniel Haus, and Gaetan de Menten. and contributors. SQLAlchemy is a trademark of Michael Bayer.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use,

copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

PyQt License

Qt License

Camelot's Documentation Copyright

Copyright (C) 2007-2012 Conceptive Engineering bvba. All rights reserved. www.conceptive.be / project-camelot@conceptive.be

This file is part of the Camelot Library.

This file may be used under the terms of the GNU General Public License version 2.0 as published by the Free Software Foundation and appearing in the file LICENSE.GPL included in the packaging of this file. Please review the following information to ensure GNU General Public Licensing requirements will be met: http://www.trolltech.com/products/qt/opensource.html

If you are unsure which license is appropriate for your use, please review the following information: http://www.trolltech.com/products/qt/licensing.html or contact project-camelot@conceptive.be.

This file is provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

For use of this library in commercial applications, please contact project-camelot@conceptive.be

7.2.2 Camelot, Qt, PyQt Licenses

Camelot License

Camelot is Copyright (C) 2007-2013 Conceptive Engineering bvba. www.conceptive.be / info@conceptive.be

You may use, distribute and copy Camelot under the terms of GNU General Public License version 2, which is displayed below.

A commercial license can be obtained from Conceptive Engineering bvba Please mail to info@conceptive.be for inquiries.

GNU GENERAL PUBLIC LICENSE Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Lesser General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

7.2.3 GNU GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. **You may modify your copy or copies of the Program or any portion** of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

    a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

    b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

    c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

    a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

    b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

    c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. **You may not copy, modify, sublicense, or distribute the Program** except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. **You are not required to accept this License, since you have not** signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

   6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

   7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

   8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the

original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this.

Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.> Copyright (C) <year> <name of author>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) year name of author Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items–whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the program 'Gnomovision' (which makes passes at compilers) written by James Hacker.
>
> <signature of Ty Coon>, 1 April 1989 Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License.

Large parts of the source in camelot/core/orm/ are based on the Elixir library (http://elixir.ematia.de).

The Elixir library was released under the MIT license, with the following copyright notice :

This is the MIT license: http://www.opensource.org/licenses/mit-license.php

Copyright (c) 2007, 2008 Jonathan LaCour, Daniel Haus, and Gaetan de Menten. and contributors. SQLAlchemy is a trademark of Michael Bayer.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

PyQt License

Qt License

7.2.4 Camelot's Documentation Copyright

Copyright (C) 2007-2012 Conceptive Engineering bvba. All rights reserved. www.conceptive.be / project-camelot@conceptive.be

This file is part of the Camelot Library.

This file may be used under the terms of the GNU General Public License version 2.0 as published by the Free Software Foundation and appearing in the file LICENSE.GPL included in the packaging of this file. Please review the following information to ensure GNU General Public Licensing requirements will be met: http://www.trolltech.com/products/qt/opensource.html

If you are unsure which license is appropriate for your use, please review the following information: http://www.trolltech.com/products/qt/licensing.html or contact project-camelot@conceptive.be.

This file is provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

For use of this library in commercial applications, please contact project-camelot@conceptive.be