Plomino Documentation

Release 1.17

Eric BREHAULT

Jul 11, 2017

Contents

Author Eric Brehault <eric.brehault@makina-corpus.com>

Contributor Jean Jordaan <jean.jordaan@gmail.com>

Contact eric.brehault@makina-corpus.com

Contents:

CHAPTER 1

About this Document

This document is aimed at Plomino application designers and managers.

This document is published under the Creative Commons by-sa-nc licence.

You are free:

- to **Share**: to copy, distribute and transmit the work
- to **Remix**: to adapt the work

under the following conditions:

- **Attribution**. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- **Non commercial**. You may not use this work for commercial purposes.
- **Share Alike**. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

CHAPTER 2

Introduction

The main objective of Plomino is to provide the ability to build business-specific applications in Plone without requiring development of Plone extension products.

Plomino allows you to create forms, to use those forms to view or edit structured contents, to filter and list those contents, to perform search requests, to add business-specific features and to automate complex processing – all entirely through the Plone web interface.

Note: Plomino is widely inspired by the IBM Lotus Domino (tm) commercial product, it reproduces its main concepts and features, and it uses its terminology (which sometimes overlaps with the Plone terminology).

Plomino is used in deployments with over 50 000 documents. Users include the UN, European banks and local government organizations.

Most Plomino users are Plone users who are not Plone developers (sometime just beginners, and sometimes experienced integrators), who, once they have built a nice website with Plone, find that specific features are not available using the standard Plone modules.

A smaller number are actual Plone developers who appreciate Plomino because it is extremely flexible or because they want their customers to be more autonomous once they have delivered their work.

Plomino derives various benefits from existing as a Plone add-on.

In the first place, Plone provides a wonderful framework, including key components for Plomino (`ZCatalog`, Zope security, PythonScripts, ...), and Plone also provides very useful features (CMS features, user management, skinning, etc.).

But the main advantage is the pluggability of Plone. Plone is the only major framework which provides real pluggability. This is a wonderful advantage because it allows Plomino to benefit from excellent Plone products very easily.

Pluggability vs. extensibility

For more on this topic, see this excellent post by Chris McDonough on Pyramid's extensibility, along with Paul Everitt's commentary. Eric Bréhault (in French) provides a wider perspective.

CHAPTER 3

Background and highlevel overview

Plomino is an interesting extension of Plone.

In a time when Plone is narrowing its focus on document-based CMS, Plomino offers in-Plone application-building.

In a time when through-the-web, *ZODB*-resident scripting has fallen out of favour, Plomino does everything through the web.

It has an interesting ancestry too: the name and some core ideas hark back to Lotus Domino (https://en.wikipedia.org/wiki/IBM_Lotus_Domino), which besides a lot of other things was a form-based application builder backed by an object database, from the early 90's. As such, in some respects it's a conceptual forerunner of Zope.

Plomino trades a number of core Plone concepts for flexibility and simplicity. In the first place, it offers only one type of content: a generic *Document* type. Secondly, it eschews containment for documents, using Plone's containment system only for its own simple application structure. Consequently, Plone's cut/copy/paste operations don't make sense for Plomino documents.

A Plomino application or database is a single container which holds *Forms*, *Views*, and *Agents*. It also has a catalog, a `documents` container for all documents, and a `resources` container for script libraries. That's it.

Positioning

Plomino is a **through-the-web application builder**, hence:

- It is **not a through-the-web content-type builder** like Dexterity, which is perfect for creating custom content types that stick to the standard content management scenario. Plomino allows to implement any custom scenarios.
- It is **not just a form generator** like PloneFormGen, as Plomino provides all the services (data storage, search, automation, import/export, etc.) to build an entire application. Regarding form generation itself, a major difference with PloneFormGen is that Plomino allows to edit the form layout entirely, while PloneFormGen uses a fixed pre-defined form layout.

A closer look

Forms contain *Fields* and *Actions*, and *Views* contain *Columns* and *Actions*.

When you start to build a Plomino application, you normally start by adding Forms. When you add a form, the creation page resembles a normal Plone Page, with a large richtext edit field. Here, you simply type out the layout of the form. Add tables, images, explanatory text, whatever you need. The one departure: you also create *fields*, *actions*, *hide-whens*, *subforms*, *accordions*, and *cache-zones* on the layout using Plomino-specific TinyMCE buttons.

This allows very quick prototyping of forms, and it broadens participation in form design. This is not equally valuable for all applications, or for all stages of an application's lifecycle, but it can be very useful.

As you add fields and so on, *Field* and *Action* instances are added to the Form. These are matched with layout elements according to id: the ids match elements on the layout, and the widget of the field or action is substituted for the placeholder on the layout when the form is viewed.

When viewing the form, a user can fill in the field widgets and submit the form using the default *Save* action. At this point, a *Document* is created containing *Item*s that correspond to the *Field*s on the *Form*.

Follow me closely here: if you create a *Book* **form** with *Title* and *Author* **fields**, this will create **documents** with *Title* and *Author* **items**. Another form, say *Trip*, may create documents with items like *Departure*, *Destination* and *Passenger*. Documents are simply generic bags of items. They are both created and viewed using forms, that render the items found on the document using the corresponding fields on the form. (Note that a given form may not have fields for all the items on the document, and there may be fields that do not correspond to items but that render values based on other items or other documents.)

With Plomino, you have to build the additional structures you need using documents and items as building blocks.

While creating documents, it may be useful to think of a Form in terms of a rubber stamp. When you use it to create a document, it stamps its items on that document, at that moment. If you change the Form afterwards, the items on the documents created previously will still be the same: you may need to re-save documents with the latest version of the form if you need their items to be updated.

While viewing documents, you are also using forms. At this point it's more useful to think of a Form in terms of a template or mask: the form will render those items that correspond to its fields (there may be more items than fields; these may be ignored, or the formula of one field may look at multiple items).

When you use a Form to create or edit a document, it stores its name in a `Form` item on the document, so you could grab all books by looking for the documents where the `Form` item is `Book`. However, Plomino doesn't require that you always use the `Book` form for editing those documents. If you added a `CatalogBook` form with fields like `Dewey` and `ISBN number`, for the use of users doing cataloging, and go over the book documents using this form, their `Form` items will change to `CatalogBook`. Therefore one common pattern is to include a `doctype` field on forms used to create documents (if, indeed, your Plomino application requires the concept of different types of documents).

Similarly you could include an item referencing a `parent` document if you wanted to mimic containment, but this is only one possible way of structuring your data.

Grouping documents

Forms are built around individual Documents. For dealing with Documents in aggregate, Plomino offers *Views*. The documents in a view are *all the documents for which the selection formula (Python Script) on the View evaluates as ‘‘True‘‘*. Views contain *Columns*, that are calculated for each matching document. They often correspond to items on documents, but can be any value returned by a formula. That is, each record in a view corresponds to a Document, but the values of columns in the record need not come from that Document.

Views are updated as documents are created or edited, but depending on the formula and the number of documents, views can be expensive to refresh from scratch.

Besides grouping documents, views are also useful for browsing purposes. They allow paging and filtering, and can evaluate a formula to determine which Form should be used for viewing documents opened from the view (that is, a view that lists books for lending could show documents using a *Checkout* form, while a view that lists books with incomplete metadata could use the *CatalogBook* form).

Security

- All the normal Plone roles and permissions pertain to Plomino.
- In addition, Plomino offers a hierarchy of roles that govern management of the application, creation and editing of Forms and other design aspects, creating and editing documents using the supplied forms, and accessing the database.
- Finally, Plomino allows creation of user-defined roles that can be assigned to Plone principals, and need to be checked for at application-level in the Plomino application.

As such, security is to some extent leaky, depending on application authors to remember the appropriate checks in all relevant forms. Also, the form to be used for rendering a document can be passed as an URL parameter, so someone could sneak a look at a document using a form that you didn’t intend, as can form values, and various other API games. This can be mitigated by factoring out certain checks to a common script library and including them in all forms, but I think you get the point — Plomino does not chase the grail of a bulletproof environment. You need to think about what is *enough* security, and not deploy Plomino applications with data inappropriate to the context (i.e. deploy applications with sensitive data to closed groups).

Barely-repeatable processes, workflow

There are countless cases of people, businesses or projects switching bug tracking systems to find one that fits their way of working. And a bug tracking system is a relatively simple domain! Most processes are much more complicated. Does this really make sense? A bug tracking system includes implementation choices and policies regarding database backend, templating mechanism, authentication sources, and so forth and so on, in addition to the business rules of bug tracking. It’s a shame that everything else has to change if you all you really want to change are the business rules.

Any application deployed in a real-world environment ends up having to deal with local variations, transient changes, emerging requirements, and having the business change in response to the application being implemented.

Of the various ways in which to confront this reality, one method is to use an architecture that provides simple building blocks. The architecture can remain stable across deployments and evolve in a controlled fashion, while the various deployments of the application can be tweaked in place, branching and diverging if needed.

This is especially true for Plomino, which is meant for quickly creating solutions where exhaustively analysing and modeling the domain is not justified; or indeed, where a Plomino solution is instrumental in building up the business knowledge necessary to realistically model a good solution, while incidentally getting work done.

This is a powerful motivation of the “dirty” mixing of content and code in the database.

Workflow

One way of addressing workflow needs in Plomino is to create a script library which computes the form which should be used based on the context (what is being viewed by whom). However Plomino itself doesn’t offer building blocks to make building workflows easy and consistent.

This makes associating security with workflow states more arduous than ideal.

Use cases

Use cases:

- simple form-based data capture.
- mini-apps that manipulate Plone content.
- selfcontained apps.
- replicate forms/data to other instances.
- pull/integrate data from other sources.

Plomino has different sweet spots. One of the quickest is simple form-based data capture. On this level, it is PloneFormGen's more free-spirited cousin.

It can also be used to manipulate Plone content, similarly to *Content Rules*, but again, it's easier to script case-by-case variations from Plomino than using Rules. This is a good case for Plomino micro-apps consisting only of a couple of forms with some scripts to drive Plone, e.g. pre-populating an event folder with Event, NewsItem, and PR announcements.

Once the bug has bitten, it's also very tempting to build entire self-contained applications in Plomino. In some cases this makes sense (for example, Plomino data and applications can be synced between Plone instances, so if you need (parts of) your application to be synced, it has to stay in Plomino), but the goal should always be to build as little as possible. For example, it would be a pity to build a bug tracker in Plomino.

Regarding the replication use cases: imagine a library environment. The forms for browsing books are synced to the public servers, but the forms for editing the catalog are kept on the librarians' servers. Or imagine a business with different branches. The data from each branch is synced to the head office to be aggregated, and pricelists are synced to branches.

Plomino can also function as a very easy integration point with legacy or third-party systems. Just arrange to push CSV to the URL of a Plomino view, or for another service to pull CSV from a Plomino view (or form or agent, depending on your needs), and complete the integration using Plomino Forms.

Digging deeper

Plomino looks nice and simple at first glance, but it allows you to get yourself into as much trouble as you like ;-)

It is conceptually quite simple, and applications are fully defined by the XML export. The core Plomino concepts could be re-implemented on Dexterity or Pyramid or Django without too much trouble. Living in a CMS has its advantages, however. The Zope and Plone APIs make a lot of power available.

Building pages

It is easy to think of Plomino in terms of simple forms-based data capture. However forms can have conditional sections, and can contain sub-forms. In addition, fields can return the rendered HTML of other forms; for example, in the `Milestones` field on a `Project` document you could look up and iterate over all the associated `Milestone` documents, get each one to render itself using an appropriate form, and include the HTML in the `Project` view. You could even return arbitrary javascript to be executed upon rendering of a form. So though you can write forms simply as richtext documents, you are also free to compute any HTML you need. For this, you have a number of mechanisms:

render documents using forms or fields, override the template used for fields or views with a template of your own, or compute exactly what you need in Python.

It is a matter of judgment at which point this becomes unmanageable. It can allow a quicker turnaround than a Python-product-based approach, but without discipline it can result in a hard-to-understand mess.

Application export and versioning

Some of the drawbacks of old-style through-the-web coding in Zope include:

- it's hard to distinguish between application and data;
- it's easy to lose track of application elements among nested folders with acquisition in play;
- it's hard to version the application.

These are mitigated in Plomino in various ways:

- A Plomino application consists of a single container with design elements (forms, views, agents), and a `resources` subfolder with scripts, templates, images, and other collateral.
- The application can be exported to XML files. The ordering and formatting of the XML is consistent and can be usefully versioned. The XML files can be imported to update an instance to a particular version of an application.

Data migration

As mentioned before, forms and documents are not tightly coupled. It's quite easy to end up with a mix of documents from the time before books had a `Translator` item and later documents that do have that item and others.

In order to deal with this, sometimes all that is needed is to code defensively. Instead of assuming that all documents will have a `Translator` item, show a default value if they don't. However if it is necessary for the item to exist, the documents need to be updated. Various approaches are possible: in the simplest case, just call the save() method on all documents. In more complicated scenarios, documents may need to be saved using specific forms or by a user with a specific role. This can be dealt with by creating a Plomino *Agent* which does the required migration.

Once there are a lot of documents, re-saving all necessary documents can take a long time. For this reason, as with all long-running Zope tasks, it's best to kick off the migration on a ZEO client set aside for jobs like this.

Caveats

A quick list of ways to make life difficult for yourself:

- Change the field type after you already have documents with items of the original type (e.g. you used to be creating strings, but now you're creating dates).
- Store complex values as items (like arrays with inconsistent formats including CSV strings).
- Store derived fields that are not computed for display (once you do this, you have to worry about keeping derived fields current when editing the reference documents).
- Have a field called "Amount" in both forms "B" & "C", both used to show doc "D", but the definition of the field on "B" is incompatible with the field on "C" (e.g. in the one case it's an integer value, and in the other it's a currency-formatted string. (This could happen if you forget to update both forms and migrate existing documents.)

Ideas for improvement

Plomino has been conservative, preferring to remain open-ended and conceptually simple. While it could be made more sophisticated in many ways, it's easy to lose some good properties in the process, such as the ability to export and version the application in its entirety, or to easily sync design elements and documents among Plomino instances.

Functionality

That said, the current weak areas of Plomino are security, workflow, and references, as they must be implemented manually using formulas.

Regarding workflow, perhaps AlphaFlow could be resurrected, or zope.wfmc or hurry.workflow could be used. A DCWorkflow-based approach would not work, as all Plomino documents share the same type, and live in the same folder.

Currently references between documents in Plomino tend to be simplistic, consisting of storing document paths or ids as items. This makes transitive relationships or keeping track of constraints on relationships error-prone and cumbersome. On the other hand, it is robust in its simplicity. If a reference engine such as zc.relationship were used, there would be the potential for the documents to get out of sync with the relationship index due to import or sync operations.

Another wrinkle regarding relations is that Plomino documents are identified by their id, which should normally not change. By default, the id is a random key. It is possible to compute something more readable, but be careful of doing so prematurely, as it makes you worry about id collisions and the continued suitability of ids chosen at the outset. Since Plomino documents can be synced among Plomino applications, relations cannot depend on object identity.

Performance

It's easy to make a big Plomino database crawl. The code being executed is Restricted Python, and rendering a form which pulls content from many related documents can pull lots of big fat Archetypes-based objects into memory. The contents of a view is anything that evaluates `True` for the view's selection formula, which may be expensive. Not bad when done incrementally, but it can be pretty bad when refreshing the view for thousands of documents.

Plomino does provide an extension mechanism, so you can move aspects of your application to filesystem-based Python code if they are mature enough and prove to be bottlenecks.

CHAPTER 4

Installation

Prerequisites

Plomino is built on Plone, so in order to install Plomino, you first need to install Plone: go to http://plone.org, download Plone and follow the instructions.

Deploy Plomino in your Plone site

To deploy the Plomino product, you need to edit your `buildout.cfg` file and add the following in the `eggs` section:

```
eggs =
     ...
     Products.CMFPlomino
```

Then you have to run `buildout` to realize your configuration:

```
bin/buildout -N
```

This will download the latest Plomino version (and its dependencies) from the http://pypi.python.org/ repository and deploy it in your Zope instance.

Now you can restart your Zope instance and in your Plone site, go to *Site setup / Add-on products*.

Here you should see `Plomino` and `plomino.tinymce` in the list of installable products. Select them and click *Install*.

Once done, Plomino is installed, so when you are in a folder, you can add a new Plomino database using the Plone *Add new...* menu.

Deploy Plomino development version

```
$ virtualenv --no-site-packages --distribute ./venv
$ cd ./venv
$ source bin/activate
$ git clone git://github.com/plomino/Plomino.git
$ cd ./Plomino
$ python bootstrap.py
$ bin/buildout -N
```

CHAPTER 5

Concepts

Plomino database

A Plomino *application* is supported by one or more Plomino *databases*.

The Plomino database is the Plone object which contains the application data (i.e. the *Documents*), and its structure (i.e. the *Design*).

Design

The *design* of a Plomino application consists of the set of *Forms* and *Views* provided in the Plomino database.

The design defines the structure of the application, and it is created by the application designer. It differs from the documents, which are the application data, created by the users.

Forms

A *form* allows users to view and/or to edit information.

A form usually contains some fields of various types (text, date, rich text, checkbox, attached files, etc.).

The application designer designs the layout he needs for the form, and inserts the fields wherever he wants.

A form can also contain some action buttons to trigger specific processing.

Forms are not always used to create or view documents — sometimes they are used to provide specific features (see *Search forms*, and *Page forms*).

Search forms

The application designer can create specific forms dedicated to perform searches. These forms are not used to create documents, but to input the search criteria.

It allows the designer to provide more specific and more business-oriented search features than the global Plone search.

Page forms

The application designer can create page forms to build custom navigation menus, generate reports, provide portlet content, etc.

Documents

A *document* is a set of data. Data can be submitted by a user using a given *form*.

Note: a document can be created using one form and then viewed or edited using a different form. The presentation of the document is determined by the form, which renders the data items found on the document. The fields on the form need not correspond one to one with the data items stored on the document: there may be more fields, or fewer fields, or the type of field may be different. Care should be taken to maintain consistency: make sure that the form matches the document.

This mechanism allows the document rendering and the displayed action buttons to change according to different parameters (user access rights, current document state, field values, etc.).

Views

A *view* defines a collection of documents.

A view has a *selection formula* which filters the documents that the application designer wants to be displayed in the view.

A view contains *columns*. *Column* contents is computed from data stored in the documents. A Plomino *view* is like a canned search. Views are built up incrementally, and not assembled dynamically as would be the case for a catalog query. Every time a document is saved, view formulas are evaluated, and if any of them return `True`, the document is included in the view.

This has two implications:

- While it is cheap to keep the view up to date incrementally, it can be expensive to rebuild views from scratch, as this involves evaluating view formulas over all documents in the database.
- A document needs to be saved in order for its state to be reflected in views. I.e. a simple `setItem` is not enough.

Views store columns as catalog metadata. This effectively doubles the storage required for any field when it is added as a view column. It trades space for speed: the last-saved value of an expensive computed field can be obtained from the view column without having to execute the field formula again (but watch out for stale values, if the state or context of the document has changed since the last save).

Columns

Views can contain columns. The column values are stored and displayed (unless hidden) for every record that forms part of the view. A *column* may refer to a form field, in which case that field will be used to render the record value, or it may specify a **column formula**, which need not correspond to one or any field.

Column totals

Numerical columns can be added up to display column totals (the total for all the records in the view). If the column refers to a field, that field will also be used to render the total.

If desired, column totals can be dynamically computed in the browser per view batch. In order to enable this, include the following snippet in the View's *Dynamic Table Parameters*:

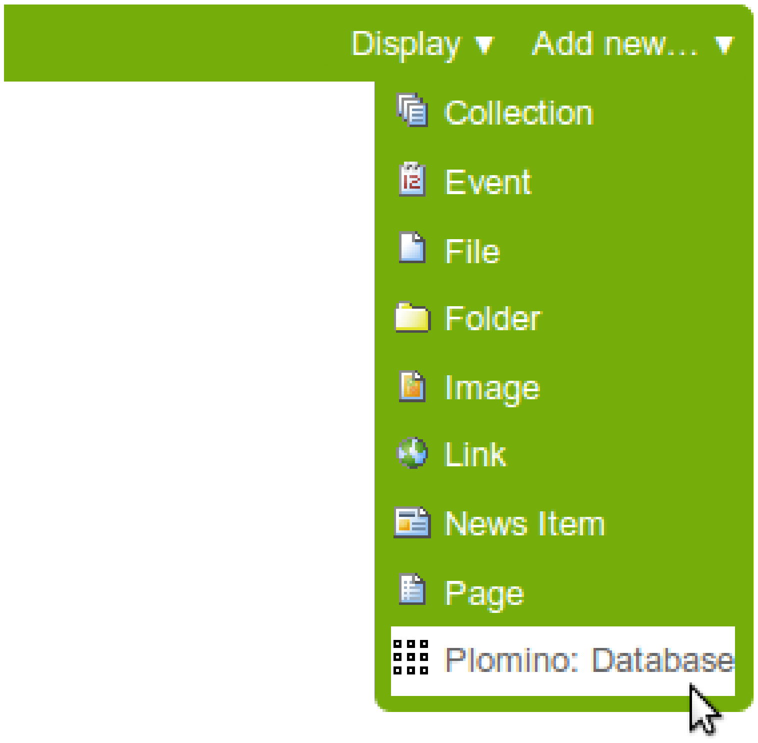
```
'fnFooterCallback': generateTableFooter,
```

CHAPTER 6

Build a simple Plomino application

Create a Plomino database

To create a Plomino database, select `Plomino: database` in the *Add item* Plone menu.

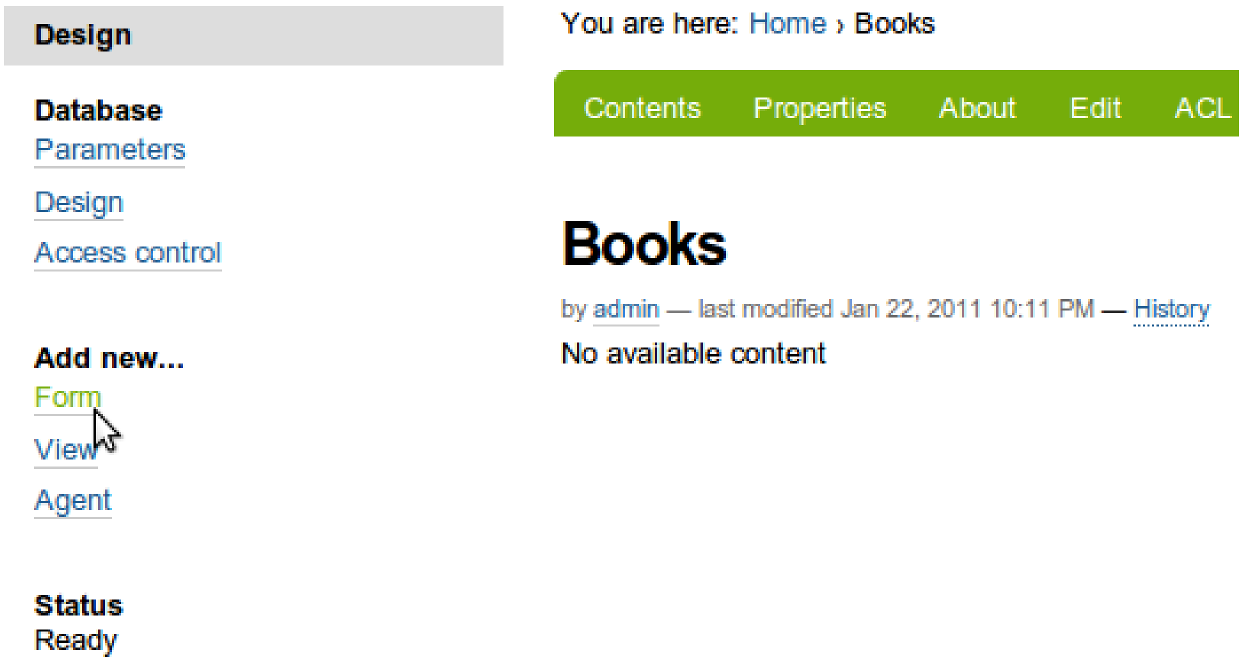


Enter a title for the database (for instance *Library*) and save it.

Add a form

Forms can be added using the Plomino Design portlet, which is usually displayed in the left-hand column, or using the Plone *Add* dropdown menu.

To add a form, click *Add new... Form* in the portlet, or select `Plomino: form` from the *Add item* Plone menu.

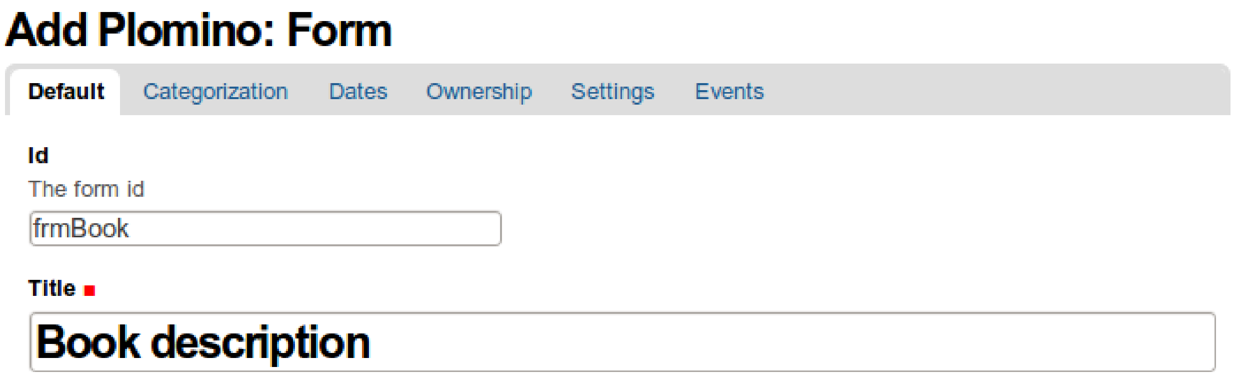


by admin — last modified Jan 22, 2011 10:11 PM — History

No available content

On the add form, enter the form id. The form id is initialized with a generated value (for instance: `plominoform.2008-01-31.9797530894`). It is preferable to replace it with a more meaningful id (for instance: `frmBook`). It is a technical identifier, so use basic characters and numbers only (blank space and special characters are forbidden).

In the *Title* field, enter the form label, which will be displayed to the users (for instance: `Book description`.



Id

The form id

frmBook

Title ■

Book description

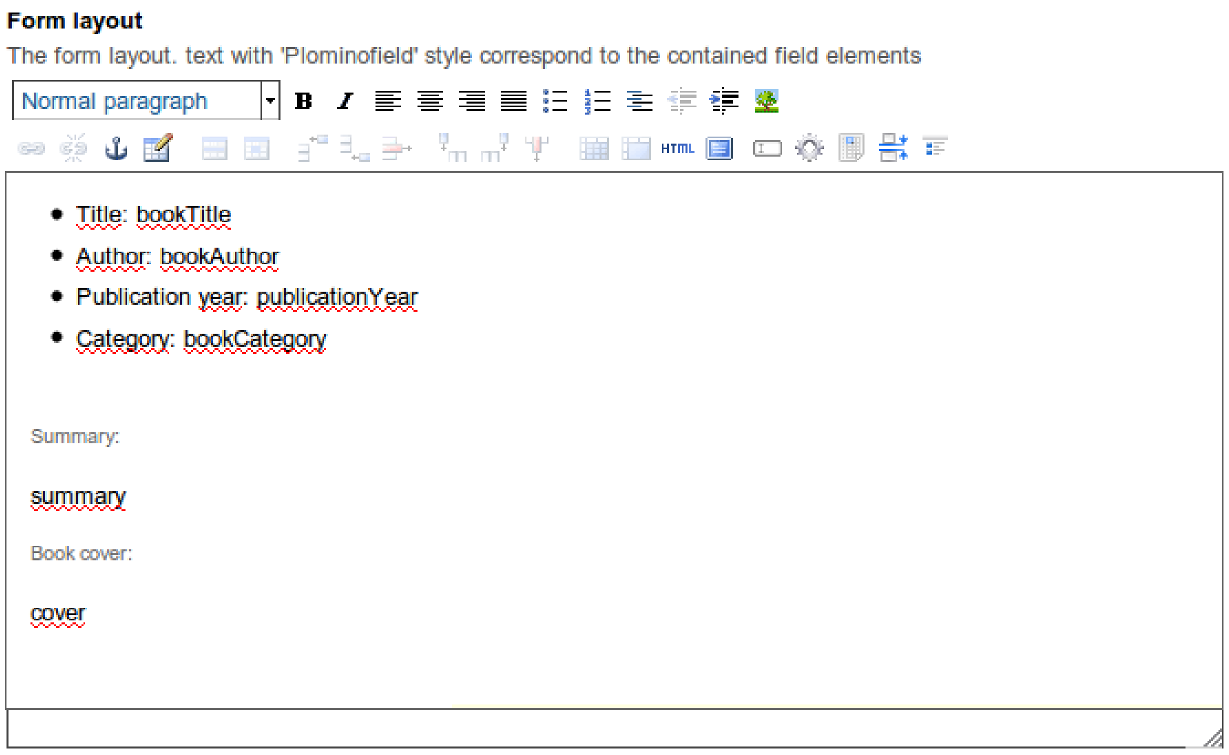
Save the form to create it (you need to save it before being able to add fields to your form).

Create the layout and add fields

Click on the *Edit* tab.

Go to the *Form layout* section which contains the TinyMCE editor. If necessary, expand the editing area by dragging the bottom-right corner, or clicking on the full-screen icon from the editor toolbar.

Create your form layout using the standard editing tools (styles, tables, etc.).



Form layout

The form layout. text with 'Plominofield' style correspond to the contained field elements

- Title: bookTitle
- Author: bookAuthor
- Publication year: publicationYear
- Category: bookCategory

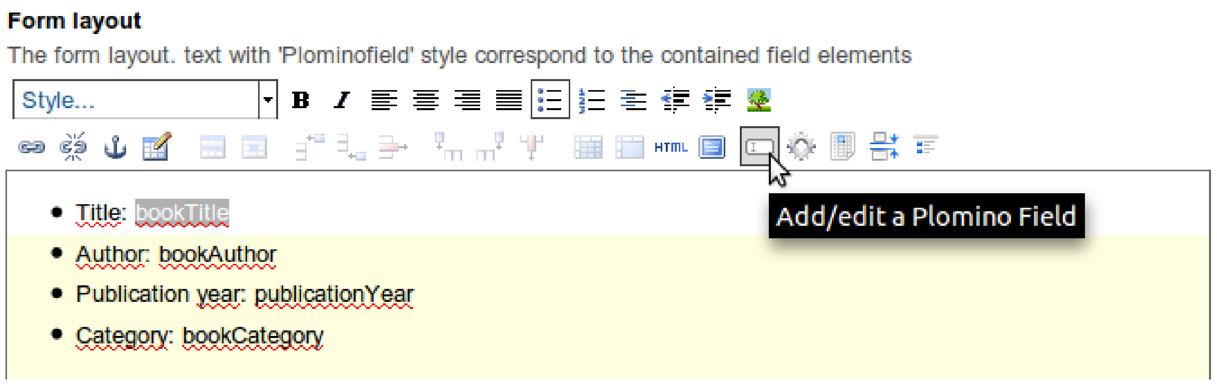
Summary:

summary

Book cover:

cover

To add a field to the layout, select some word in the layout and click on the *Add/edit Plomino field* button in the TinyMCE toolbar.

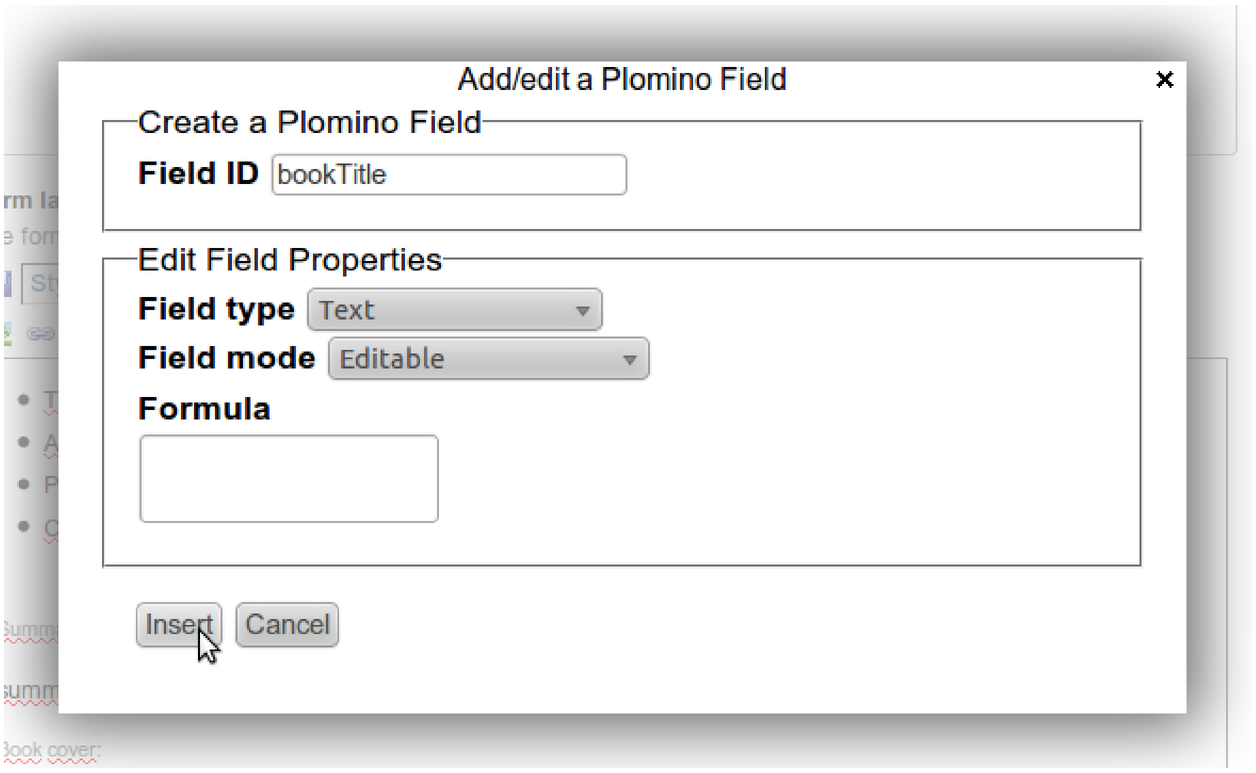


Form layout

The form layout. text with 'Plominofield' style correspond to the contained field elements

- Title: bookTitle
- Author: bookAuthor
- Publication year: publicationYear
- Category: bookCategory

The selected text will be used as the field id, and a pop-up window will allow you to enter the field main parameters:


Add/edit a Plomino Field

Create a Plomino Field

Field ID bookTitle

Edit Field Properties

Field type Text

Field mode Editable

Formula

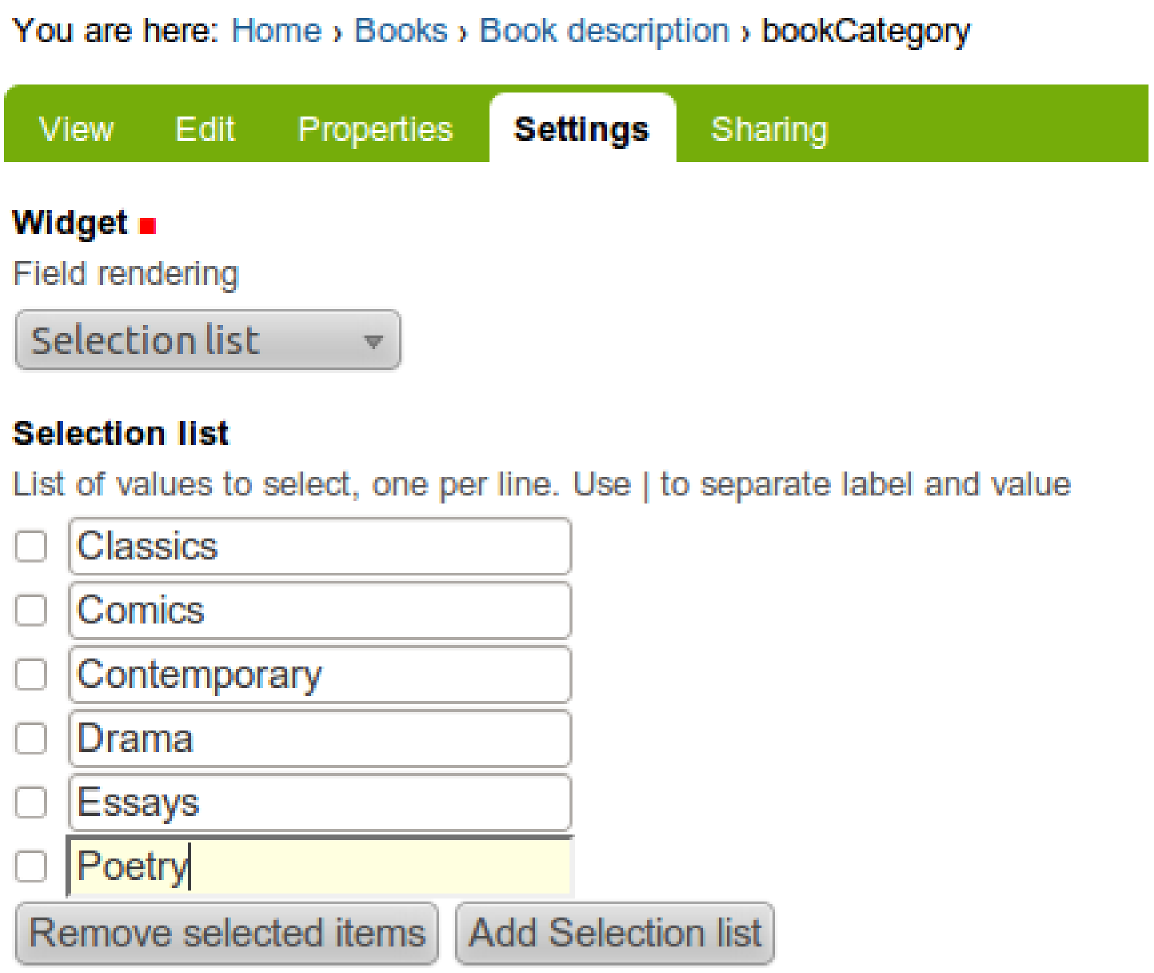
For the `bookAuthor` field, keep the default values ('Text' and 'Editable'), click *Insert* and then *Close*.

As you can see, the field is rendered with a blue dashed border in the layout.

Do the same for the following fields:

- `bookTitle`, type 'Text', 'Editable'
- `publicationYear`, type 'Number', 'Editable'
- `summary`, type 'Rich text', 'Editable'
- `cover`, type 'File attachment', 'Editable'
- `bookCategory`, choose type 'Selection list', 'Editable', but after clicking *Insert*, click on *Specific settings*.

This opens the field settings page in a new window, where you can enter the possible values for the Selection list:


You are here: Home › Books › Book description › bookCategory

Widget

Field rendering

Selection list

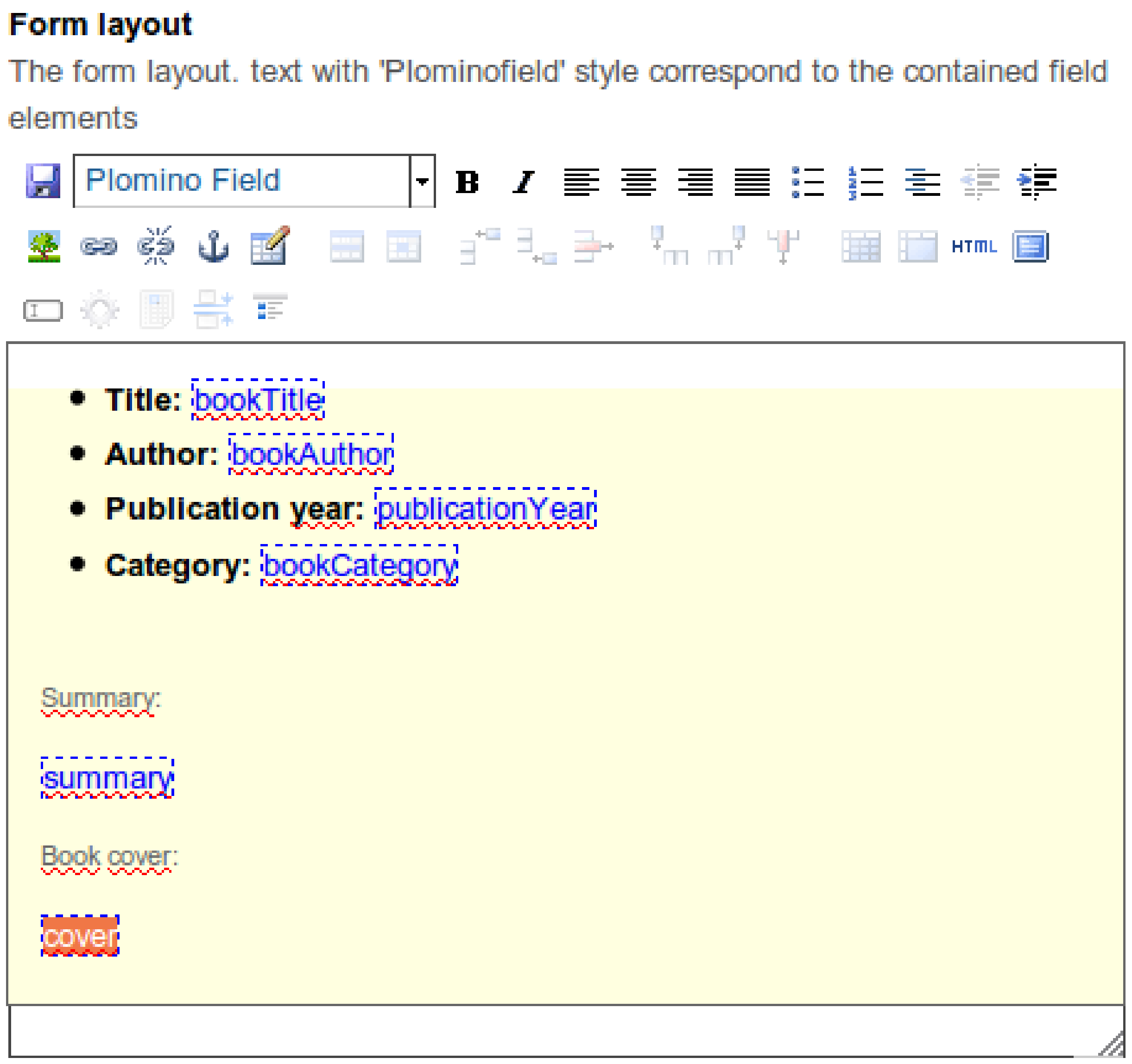
Selection list

List of values to select, one per line. Use | to separate label and value

- Classics
- Comics
- Contemporary
- Drama
- Essays
- Poetry

Click *Apply*, go back to the Form window, and close the field pop-up.

Now the form is built, and its associated fields have been created.


Form layout

The form layout. text with 'Plominofield' style correspond to the contained field elements

- **Title:** bookTitle
- **Author:** bookAuthor
- **Publication year:** publicationYear
- **Category:** bookCategory

Summary:

summary

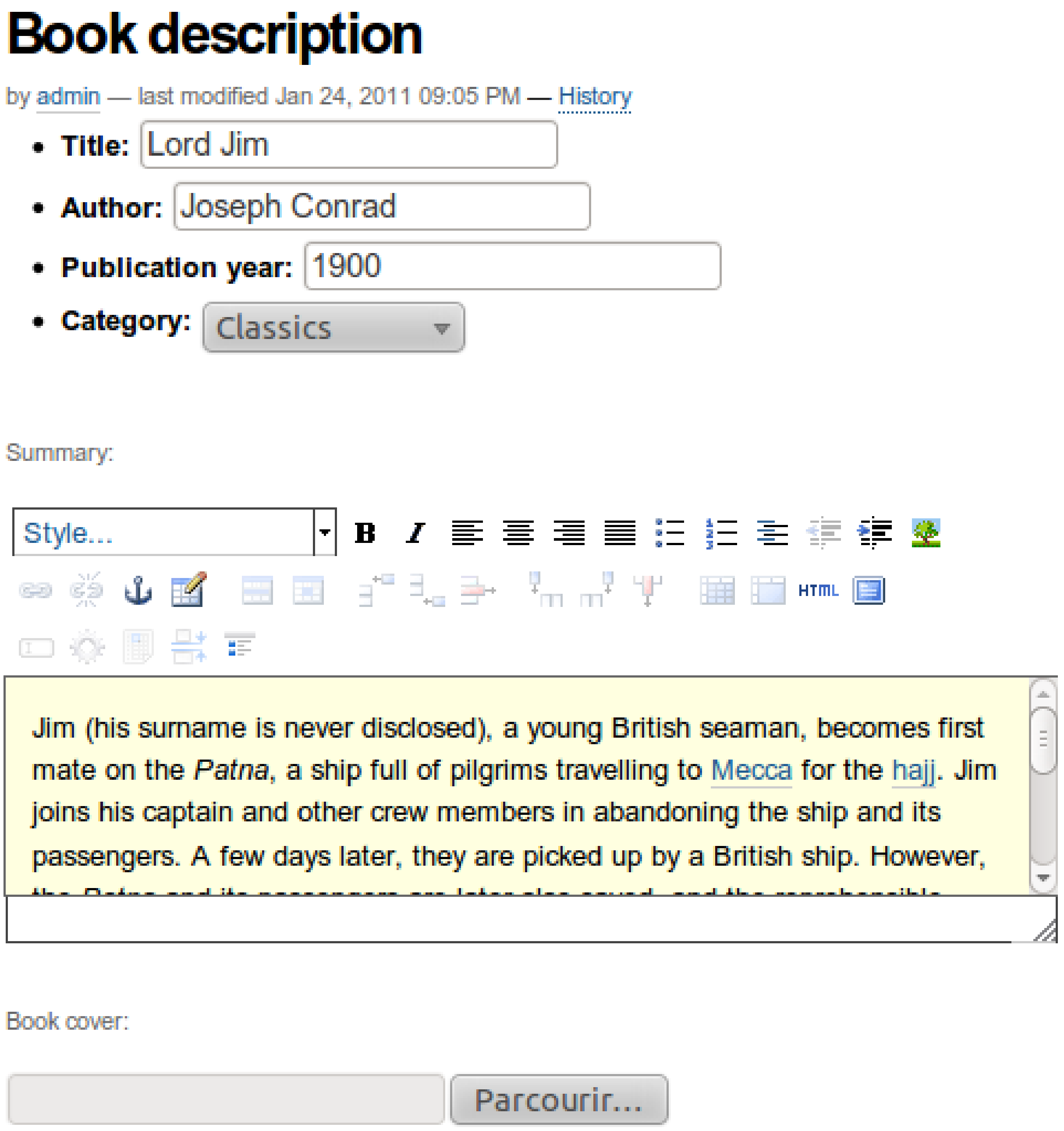
Book cover:

cover

Save the form (click the *Save* button at the bottom of the page).

Use the form

You can now use this form to create documents.


Book description

by admin — last modified Jan 24, 2011 09:05 PM — History

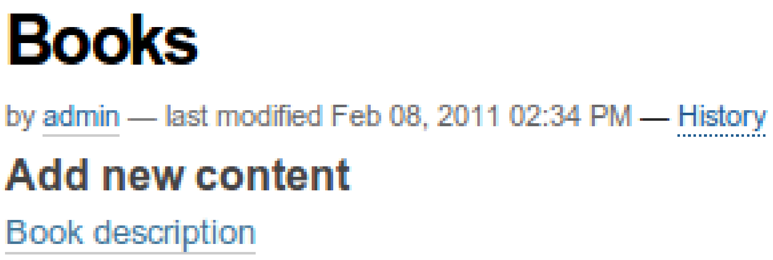
- Title: Lord Jim
- Author: Joseph Conrad
- Publication year: 1900
- Category: Classics

Summary:

Jim (his surname is never disclosed), a young British seaman, becomes first mate on the *Patna*, a ship full of pilgrims travelling to Mecca for the hajj. Jim joins his captain and other crew members in abandoning the ship and its passengers. A few days later, they are picked up by a British ship. However,

Book cover:

Go back to the *Library* database. The database welcome page now contains a link to add a new document using the `Book description` form:

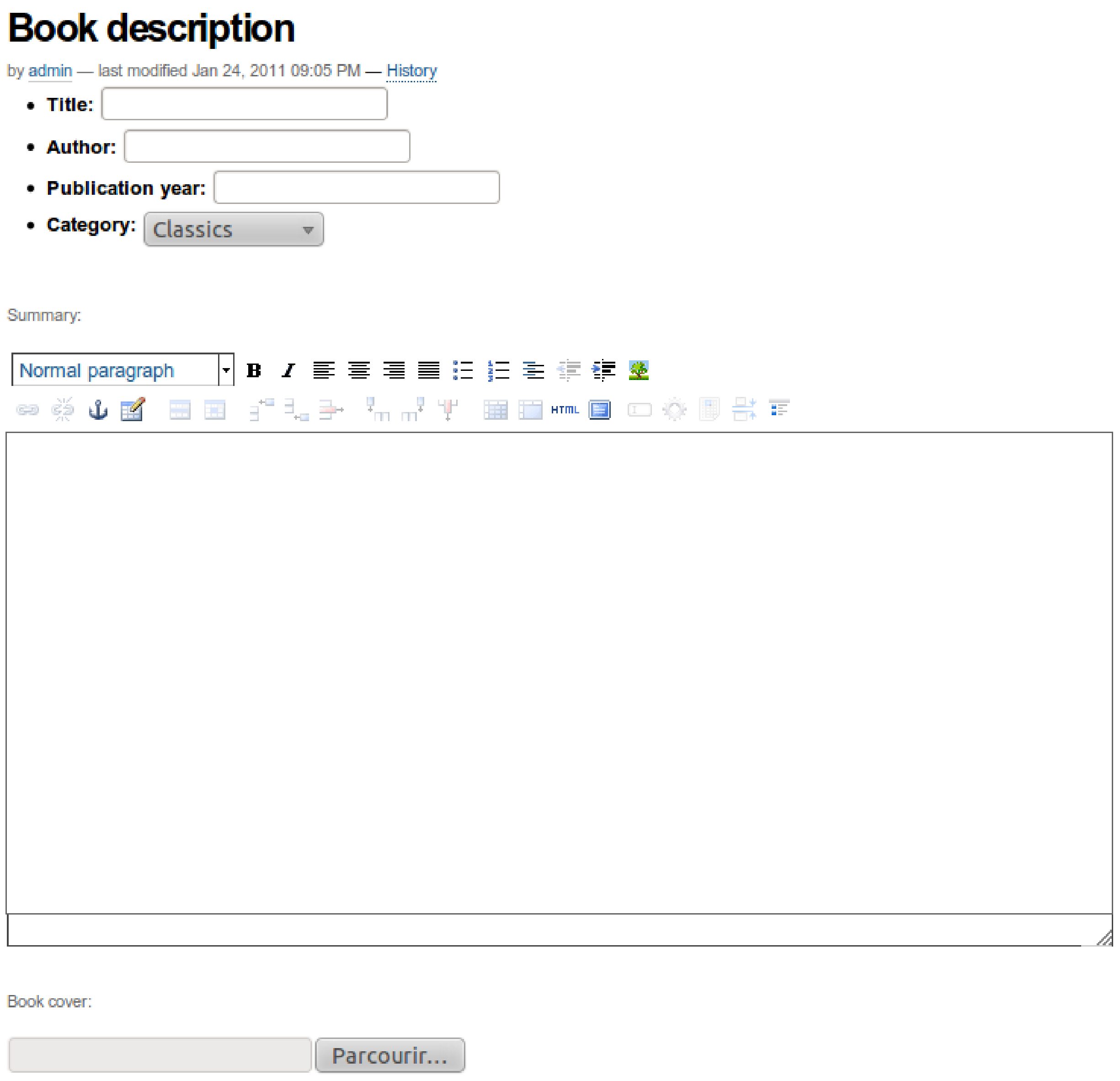

Books

by admin — last modified Feb 08, 2011 02:34 PM — History

Add new content

Book description

Click on this link, and you get the form displayed as designed in the TinyMCE editor, including the fields as they have been defined:



Book description

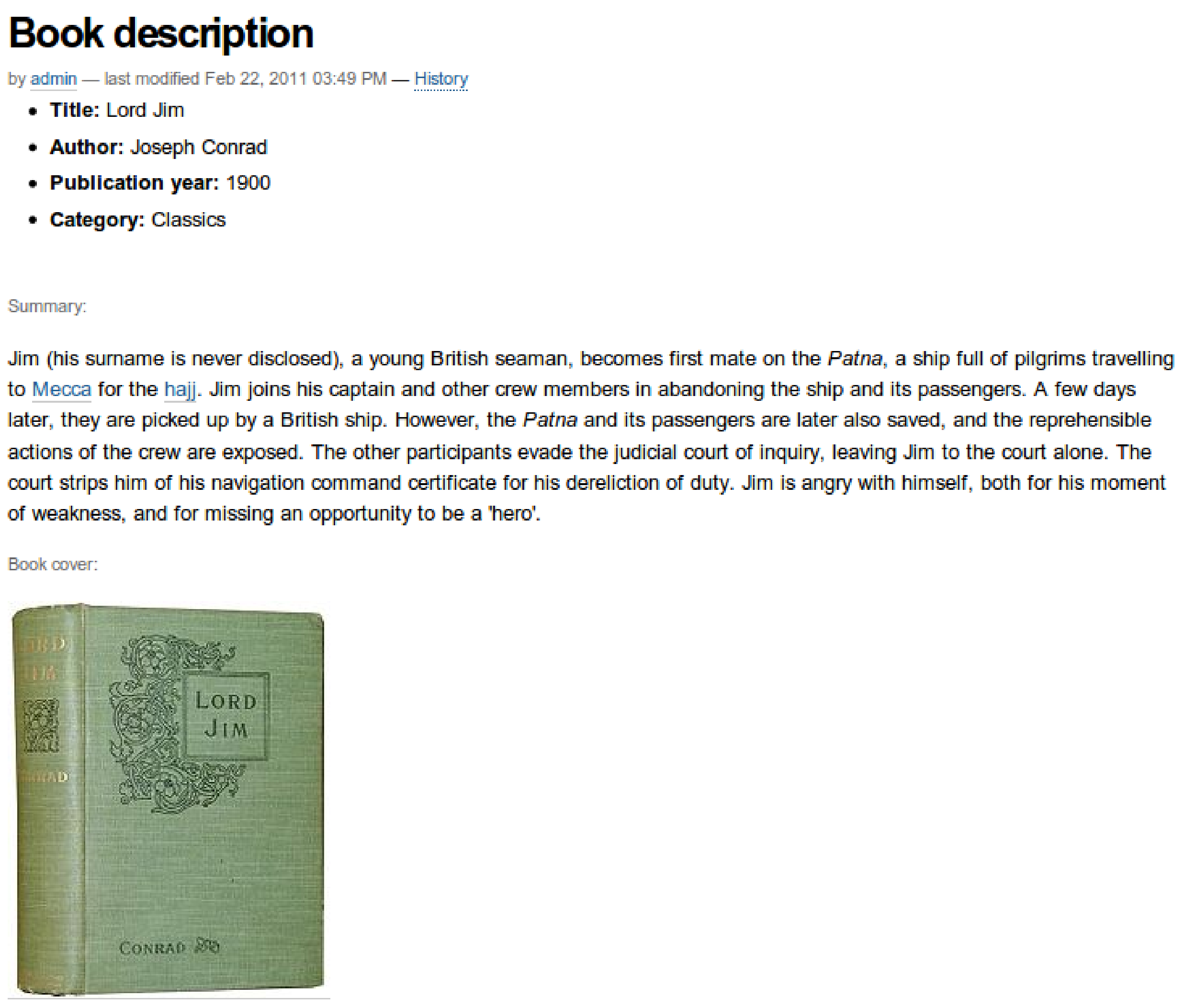
by admin — last modified Jan 24, 2011 09:05 PM — History

- **Title:**
- **Author:**
- **Publication year:**
- **Category:** Classics

Summary:

Book cover:

You can enter values and save, and a new document will be created:



Book description

by admin — last modified Feb 22, 2011 03:49 PM — History

- **Title:** Lord Jim
- **Author:** Joseph Conrad
- **Publication year:** 1900
- **Category:** Classics

Summary:

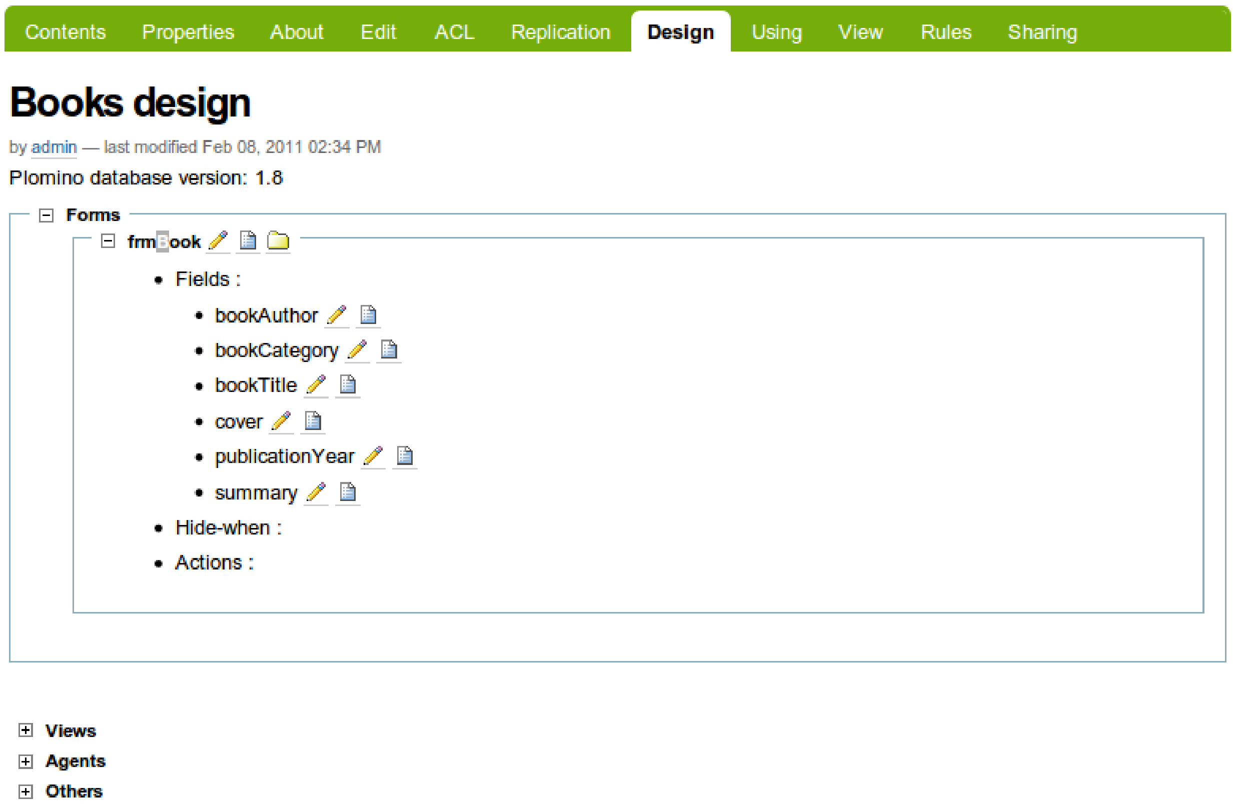
Jim (his surname is never disclosed), a young British seaman, becomes first mate on the *Patna*, a ship full of pilgrims travelling to Mecca for the hajj. Jim joins his captain and other crew members in abandoning the ship and its passengers. A few days later, they are picked up by a British ship. However, the *Patna* and its passengers are later also saved, and the reprehensible actions of the crew are exposed. The other participants evade the judicial court of inquiry, leaving Jim to the court alone. The court strips him of his navigation command certificate for his dereliction of duty. Jim is angry with himself, both for his moment of weakness, and for missing an opportunity to be a 'hero'.

Book cover:

Explore the database design

Go to the *Library* database and click the *Design* tab.

This tab displays all the design elements contained in the database:



Books design

by admin — last modified Feb 08, 2011 02:34 PM

Plomino database version: 1.8

- Forms
  - frmBook
    - Fields :
      - bookAuthor
      - bookCategory
      - bookTitle
      - cover
      - publicationYear
      - summary
    - Hide-when :
    - Actions :
- Views
- Agents
- Others

The pencil icon gives access to the corresponding object in edit mode, the page icon in read mode, and the folder icon in content mode.

Change the document title

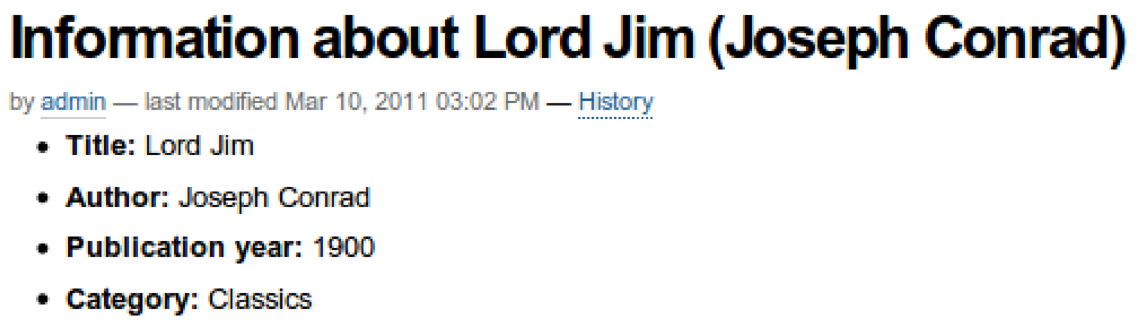
By default, all the documents created with a form have the same title as the form.

In the present case, the title is “Book description”, and it will be the title of all the documents you would create with your form.

To display a more useful title, go to the `frmBook` object, edit it, and enter the following formula in the *Document title formula* field:

```
return "Information about %s (%s)" % (
    plominoDocument.getItem('bookTitle'),
    plominoDocument.getItem('bookAuthor'))
```

Save the form, go back to the document, make a change and save it. This will trigger calculation of the title formula. Now you will see the title has been set as specified in the formula:



Infomation about Lord Jim (Joseph Conrad)

by admin — last modified Mar 10, 2011 03:02 PM — History

- **Title:** Lord Jim
- **Author:** Joseph Conrad
- **Publication year:** 1900
- **Category:** Classics

The document title is computed by a formula. All Plomino formulas are restricted Python scripts with certain variables and functions provided. In this case, the `plominoDocument` variable is used, which is the current document.

All the data items stored on the document by forms, or set using formulas, are accessible using the `getItem` API: (`plominoDocument.getItem(<field name>)`).

For more information about formulas, see Formulas below.

Change the document id

The document id is used in the URL. By default, it is an opaque random identifier (`4e219e4ffff21b9753c94a0e006e95bf` in the following):

```
http://localhost:8090/demo/books/plomino_documents/4e219e4ffff21b9753c94a0e006e95bf
```

If you want to use meaningful ids, you can define a *Document id formula*. Go to the `frmBook` object, edit it, and enter the following formula in *Document id formula*:

```
plominoDocument.bookTitle +"-"+plominoDocument.bookAuthor
```

Unlike the title, the id is computed at creation time, and it cannot be changed later. So the existing document will not use this formula even if we re-save it. But if you create a new document, you will get a id corresponding to your formula:

```
http://localhost:8090/demo/books/plomino_documents/1919-john-dospassos
```

Warning: If you use this facility, you need to take care that document ids are unique, well-formed, and resolve any issues that arise when replicating documents to other Plomino instances. Calculating your own document ids can be a considerable responsibility, depending on the requirements of your application.

Add a view

A *view* defines a collection of documents. Some views are used to present lists of documents to users, and some are used from formulas to structure the Plomino application.

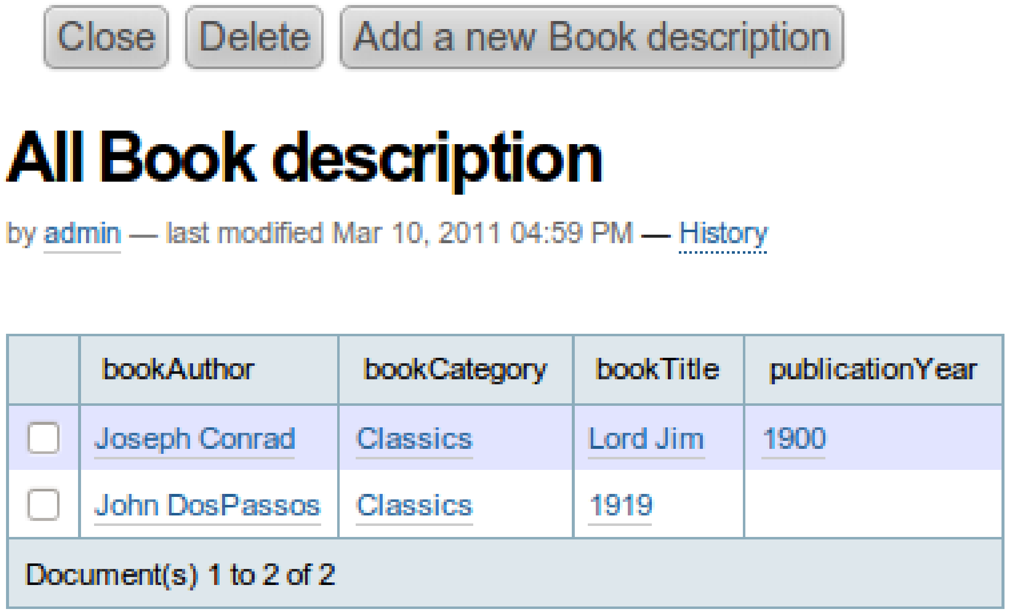
A view has a selection formula, which defines which documents form part of the view, and it usually contains some columns to display information about the matching documents. These columns may compute derived information from data items on documents, or even from values looked up from other documents, Plone objects, or other sources.

You can generate a view automatically from a form:

- Go to the `frmBook` form, and
- click on *Generate view* in the *Design* portlet on the left.

This generates a view which:

- selects all the documents that were created or last edited using the `frmBook` form,
- creates a *column* for each field on the form (file attachments and rich text fields are skipped), and it also
- inserts an *Add new* action.



All Book description

by admin — last modified Mar 10, 2011 04:59 PM — History

	bookAuthor	bookCategory	bookTitle	publicationYear
☐	Joseph Conrad	Classics	Lord Jim	1900
☐	John DosPassos	Classics	1919	

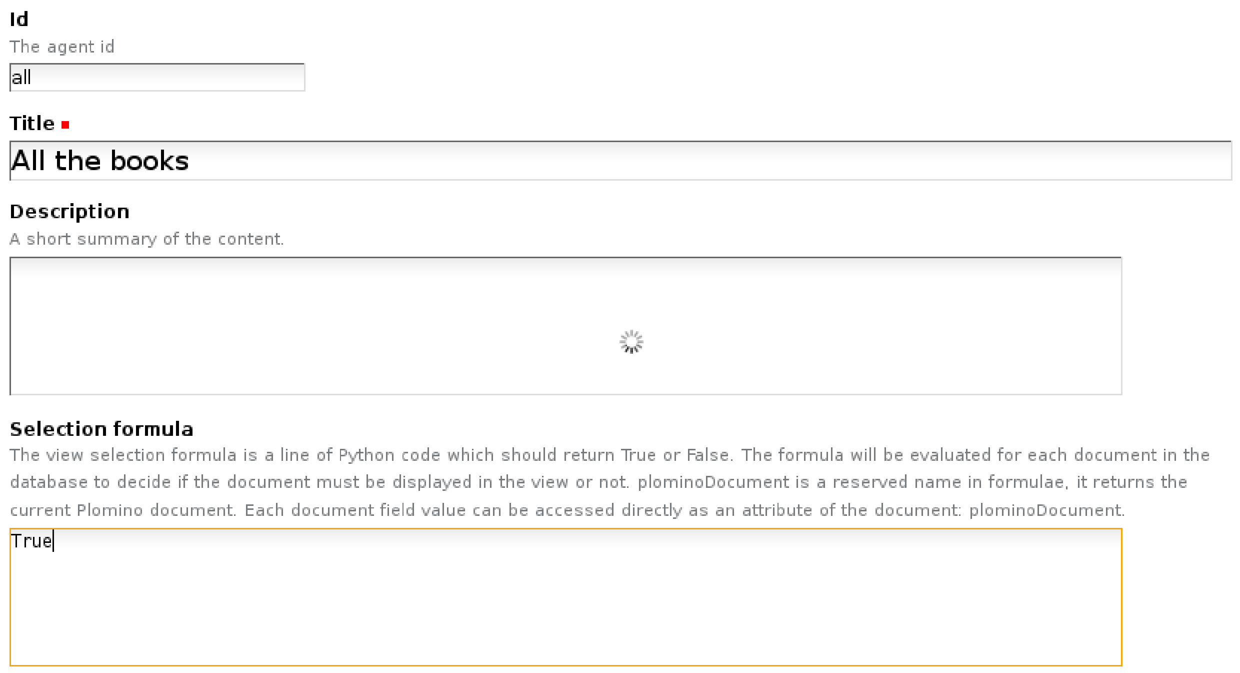
Document(s) 1 to 2 of 2

The columns can be re-ordered by drag-and-drop in the *Contents* tab. The column labels can also be changed.

Add a view manually

Go back to the *Library* database.

Select `Plomino: view` from the *Add item* Plone menu. Enter an identifier (`allBooks`) and a title ('All the books'):



Id

The agent id

all

Title

All the books

Description

A short summary of the content.

Selection formula

The view selection formula is a line of Python code which should return True or False. The formula will be evaluated for each document in the database to decide if the document must be displayed in the view or not. plominoDocument is a reserved name in formulae, it returns the current Plomino document. Each document field value can be accessed directly as an attribute of the document: plominoDocument.

True

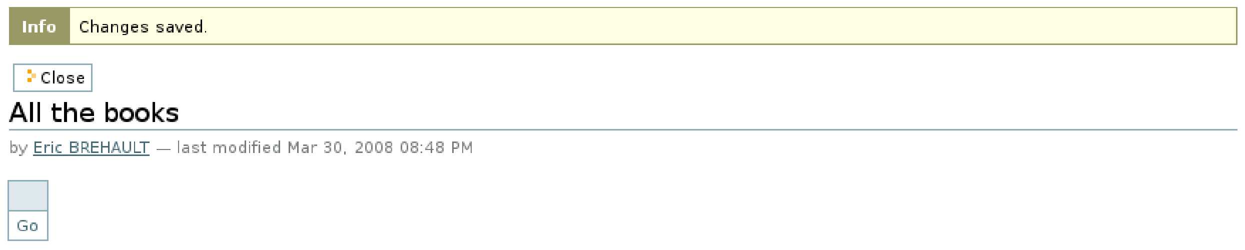
Enter a selection formula too: this formula must return `True` or `False`. It is evaluated for each document; if the returned value is `True`, the document is included in the view; if `False`, it is rejected.

Enter the following expression and hit *Save*:

```
return True
```

(this expression always returns `True`, so all the documents will be displayed).

You get the following result:



Info Changes saved.

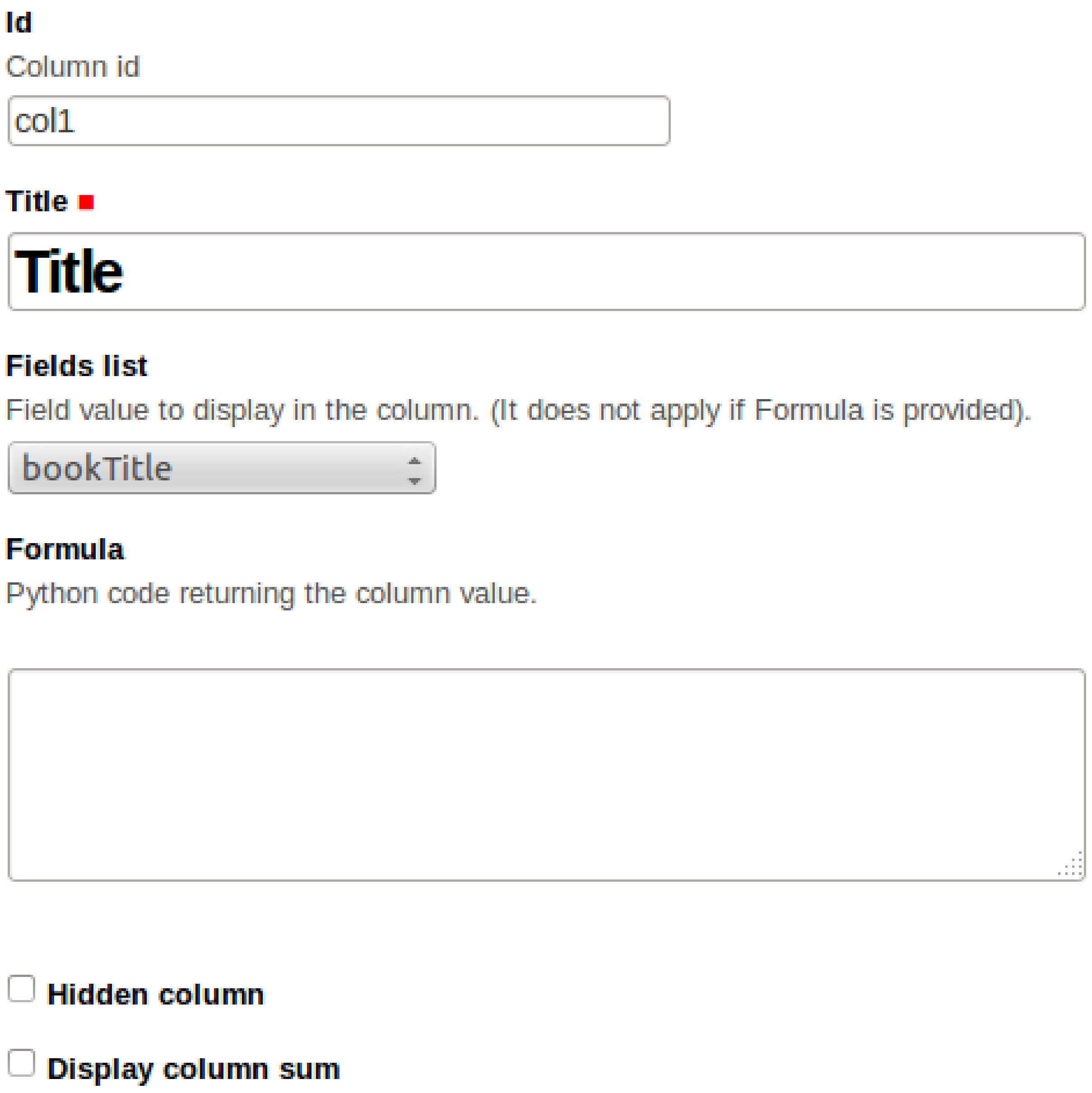
All the books

by Eric BREHAULT — last modified Mar 30, 2008 08:48 PM

We just see a link *Go* which allows us to access the document we have created. Now we need to add columns to this view.

Select `Plomino: column` from the *Add item* Plone menu.

Enter an identifier and a title, and select the field you want to display in the column.



Id

Column id

col1

Title

Title

Fields list

Field value to display in the column. (It does not apply if Formula is provided).

bookTitle

Formula

Python code returning the column value.

☐ Hidden column

☐ Display column sum

You can also enter a *formula* to compute the column value, for instance:

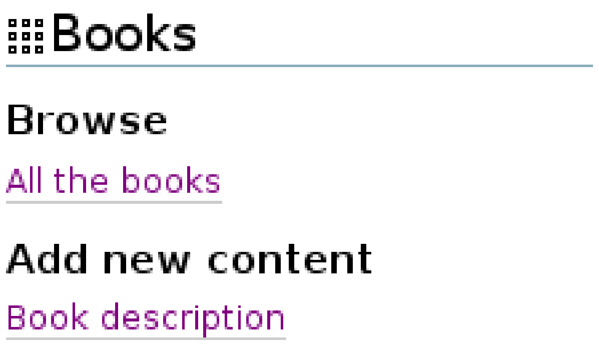
```
return plominoDocument.getItem('bookTitle').upper()
```

Warning: When you use a field as column value, the Plomino index will use the field index. So if you display this field as column in several views, it will not increase the index size. But when you create a formula, it will create a new column-specific index, so having a lot of column formulas might impact the database global performances.

Similarly, add a column to display `bookAuthor`.

Columns can be ordered by going to the view's *Contents* tab and moving the columns where needed.

If you go back to the Library database root, the view is proposed in the *Browse* section:



Books

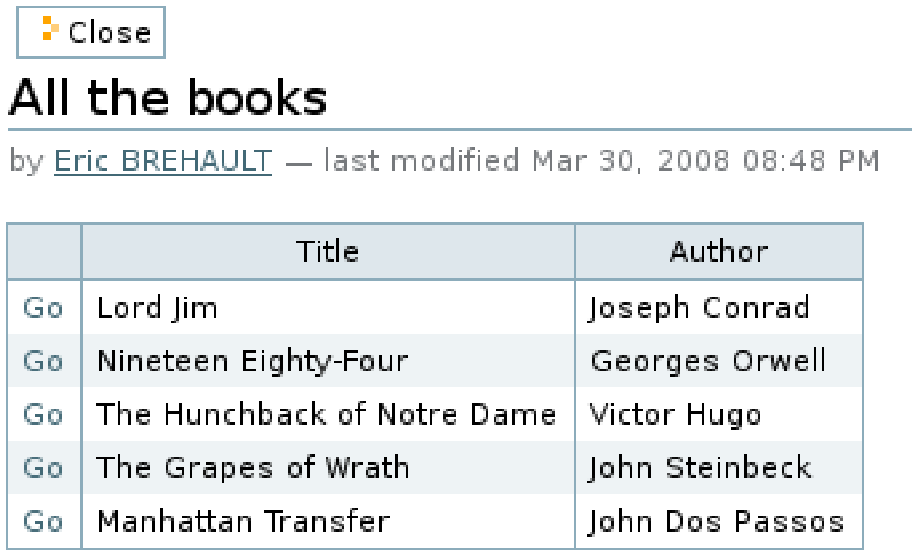
Browse

All the books

Add new content

Book description

Create more documents. When you click on the link *All the books*, the view is displayed with its 2 columns (and its new documents):



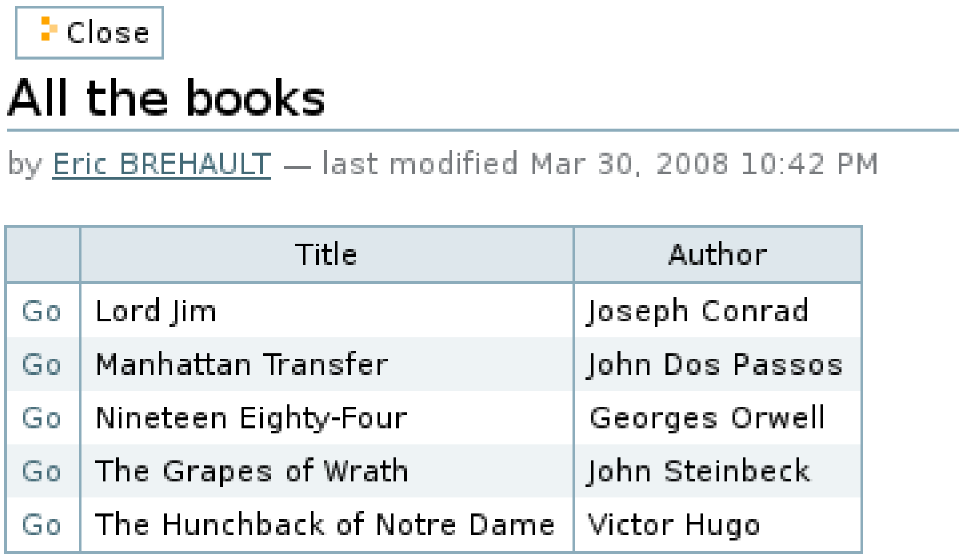
All the books

by Eric BREHAULT — last modified Mar 30, 2008 08:48 PM

	Title	Author
Go	Lord Jim	Joseph Conrad
Go	Nineteen Eighty-Four	Georges Orwell
Go	The Hunchback of Notre Dame	Victor Hugo
Go	The Grapes of Wrath	John Steinbeck
Go	Manhattan Transfer	John Dos Passos

To improve browsing of the documents, it could be useful to sort the view.

To do that, click on *Edit*, go to the *Sorting* tab and enter `col1` in the *Sorting* column, then save:



All the books

by Eric BREHAULT — last modified Mar 30, 2008 10:42 PM

	Title	Author
Go	Lord Jim	Joseph Conrad
Go	Manhattan Transfer	John Dos Passos
Go	Nineteen Eighty-Four	Georges Orwell
Go	The Grapes of Wrath	John Steinbeck
Go	The Hunchback of Notre Dame	Victor Hugo

Add more views

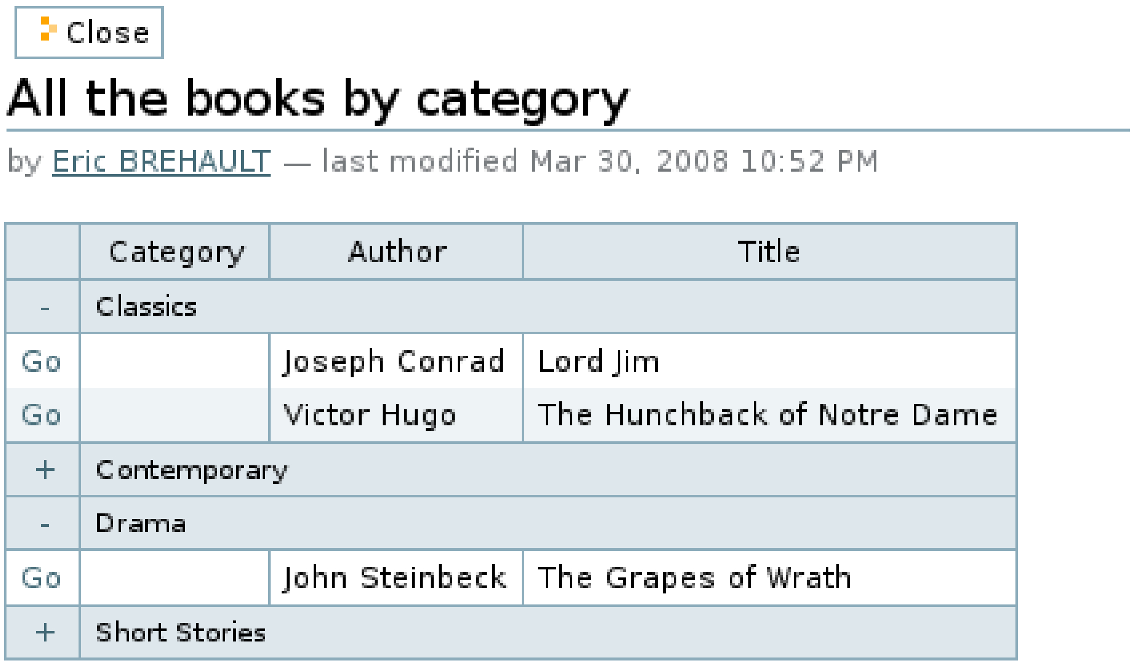
You can add as many views as necessary.

You can build views able to filter the documents; for instance if you enter the following selection formula:

```
return (plominoDocument.getItem('publicationYear') >= 1800 and
    plominoDocument.getItem('publicationYear') < 1900)
```

you will only list the XIXth century books.

You can create *categorised* views: create a view with a first column which contains the `bookCategory` field value, and select *Categorised* in the *Sorting* tab:



All the books by category

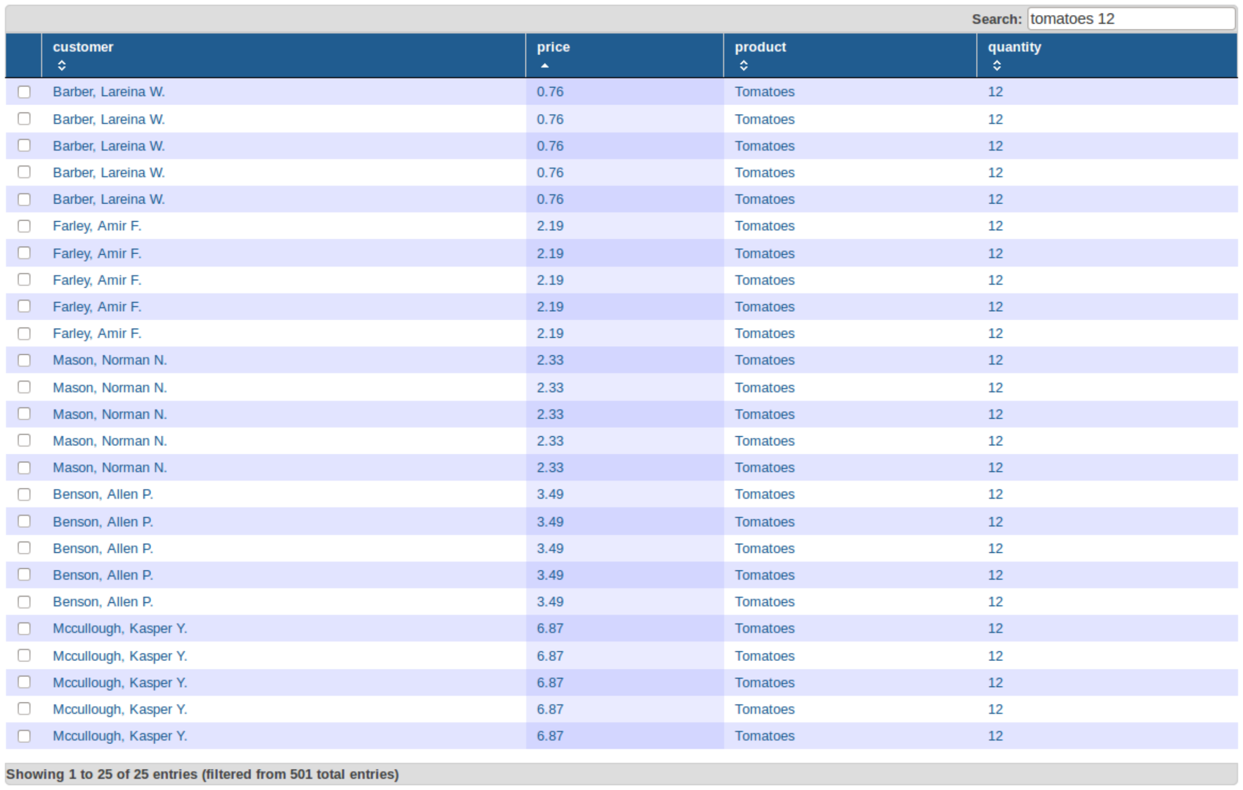
by Eric BREHAULT — last modified Mar 30, 2008 10:52 PM

	Category	Author	Title
-	Classics		
Go		Joseph Conrad	Lord Jim
Go		Victor Hugo	The Hunchback of Notre Dame
+	Contemporary		
-	Drama		
Go		John Steinbeck	The Grapes of Wrath
+	Short Stories		

Each category can be expanded or collapsed.

Dynamic view

Click on *Edit*, go to the *Parameters*, and change widget to *Dynamic table*. It renders the view using JQuery Datatables (column sorting, live filtering, ...).



Search: tomatoes 12

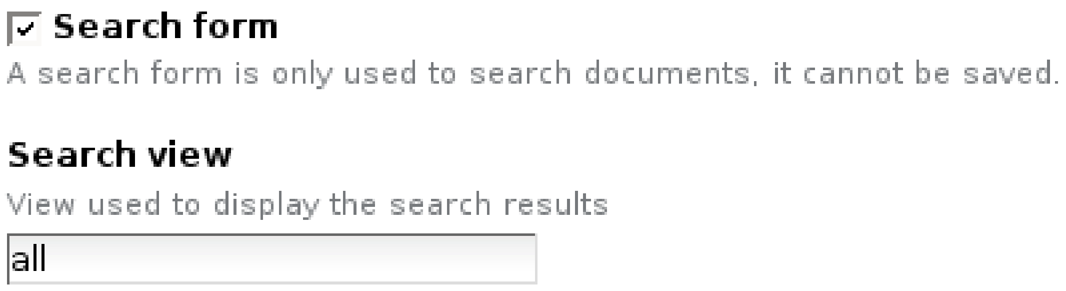
	customer	price	product	quantity
☐	Barber, Lareina W.	0.76	Tomatoes	12
☐	Barber, Lareina W.	0.76	Tomatoes	12
☐	Barber, Lareina W.	0.76	Tomatoes	12
☐	Barber, Lareina W.	0.76	Tomatoes	12
☐	Barber, Lareina W.	0.76	Tomatoes	12
☐	Farley, Amir F.	2.19	Tomatoes	12
☐	Farley, Amir F.	2.19	Tomatoes	12
☐	Farley, Amir F.	2.19	Tomatoes	12
☐	Farley, Amir F.	2.19	Tomatoes	12
☐	Farley, Amir F.	2.19	Tomatoes	12
☐	Mason, Norman N.	2.33	Tomatoes	12
☐	Mason, Norman N.	2.33	Tomatoes	12
☐	Mason, Norman N.	2.33	Tomatoes	12
☐	Mason, Norman N.	2.33	Tomatoes	12
☐	Mason, Norman N.	2.33	Tomatoes	12
☐	Benson, Allen P.	3.49	Tomatoes	12
☐	Benson, Allen P.	3.49	Tomatoes	12
☐	Benson, Allen P.	3.49	Tomatoes	12
☐	Benson, Allen P.	3.49	Tomatoes	12
☐	Benson, Allen P.	3.49	Tomatoes	12
☐	Mccullough, Kasper Y.	6.87	Tomatoes	12
☐	Mccullough, Kasper Y.	6.87	Tomatoes	12
☐	Mccullough, Kasper Y.	6.87	Tomatoes	12
☐	Mccullough, Kasper Y.	6.87	Tomatoes	12
☐	Mccullough, Kasper Y.	6.87	Tomatoes	12

Showing 1 to 25 of 25 entries (filtered from 501 total entries)

Add a search form

Create a new form named `frmSearch`, and add some fields with the same identifiers as the documents fields you want to be able to search; for instance: `bookTitle`, `bookAuthor` and `bookCategory`.

In the *Parameters* tab, select 'Search form' and enter `all` in *Search view*:



☑ Search form

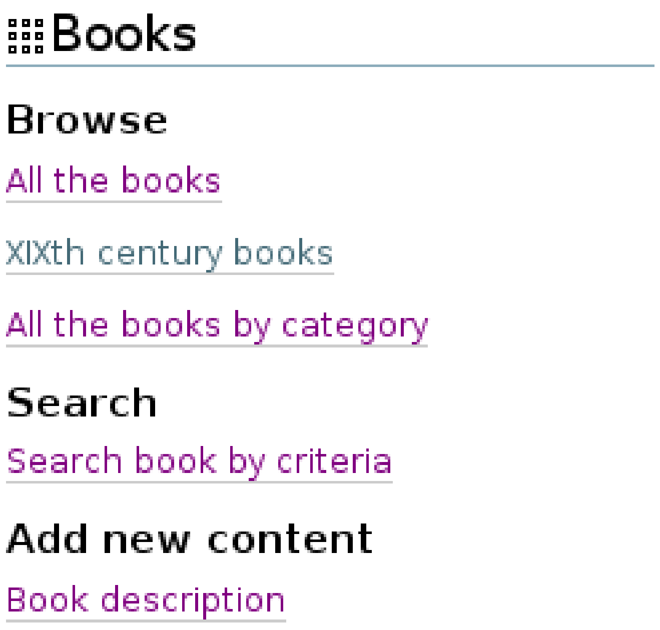
A search form is only used to search documents, it cannot be saved.

Search view

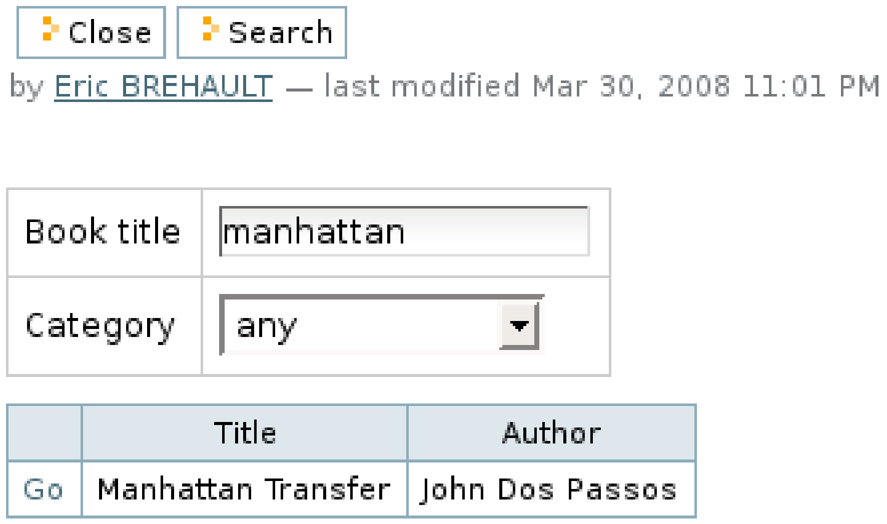
View used to display the search results

all

This form is now proposed in the *Search* section in the Library database root:



If you click on this link, you get the search form, and if you enter some criteria, the results are displayed under the form:



Book title	manhattan
Category	any

	Title	Author
Go	Manhattan Transfer	John Dos Passos

Note: the criteria are effective only if the field names match the document item names.

About and *Using* pages

Go to the *Library* database *Edit* tab. You can fill in the *About this database* section and the *Using this database* section.

Information entered here will be available under the *About* and the *Using* tabs. It allows you to offer users a page to describe the purpose of the application and another one to give a short user guide.

CHAPTER 7

Access control

Standard Plomino access rights

Plomino offers 5 standard access levels for any Plomino database:

Reader can read any document, perform searches, but cannot create new documents or modify existing ones.

Author Reader + can create new documents, and modify/delete only documents he/she has created.

Editor Author + can modify/delete any documents.

Designer Editor + can change the database design, but cannot edit formulas.

Manager Designer + can change formulas + can change the access rights.

These rights can be granted to Plone members and/or to Plone groups.

Note: in reality, Designer users could in fact edit formulas, we just hide the editing UI. So the Designer role is not a security restriction, it is just a UI behaviour useful to allow non-coder users to easily modify a form layout without breaking the Python code in the different formulas.

Generic users

Plomino handles 2 types of generic users:

Anonymous users not authenticated on the Plone site.

Authenticated any authenticated user.

The Plomino standard access rights can be applied to those 2 generic users, but an anonymous user will never be able to delete a document.

Note: as nothing can differentiate an anonymous user from another one, this rule exists to ensure that no one will delete a document created by someone else.

Roles

Some applications may need to provide, for specific users, a specific behaviour which is beyond the basic access rights mechanism we have just described.

Plomino allows you to create roles which can be applied to Plone users.

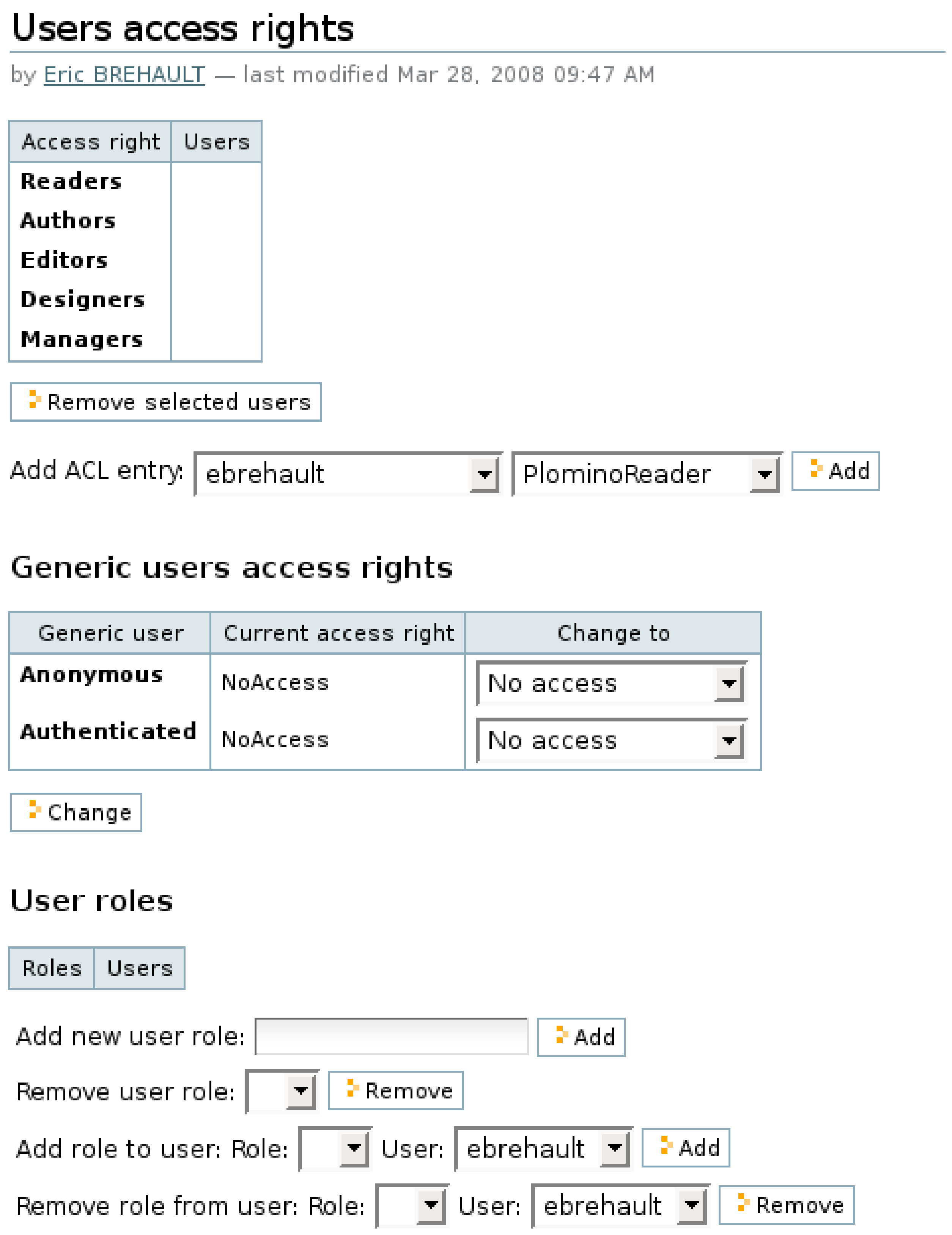
By default, a role does not grant any extra rights, but the application designer will use them as markers to enable specific behaviours in his application.

For instance, if you build a Plomino application to handle purchase requests, all the employees will be able to use the form to submit a purchase request, but in the form you would check for the `[FinancialReponsible]` role to allow access to the *Approval* section.

Note: roles are always noted with brackets.

Manage the access rights

Access rights are managed in the tab named *ACL* (Access Control List).



Users access rights

by Eric BREHAULT — last modified Mar 28, 2008 09:47 AM

Access right	Users
Readers	
Authors	
Editors	
Designers	
Managers	

Remove selected users

Add ACL entry: ebrehault PlominoReader Add

Generic users access rights

Generic user	Current access right	Change to
Anonymous	NoAccess	No access
Authenticated	NoAccess	No access

Change

User roles

Roles	Users

Add new user role: Add

Remove user role: Remove

Add role to user: Role: User: ebrehault Add

Remove role from user: Role: User: ebrehault Remove

Application-level access control

In addition to the global access rights, it may also be necessary to configure access to documents individually.

Plomino_Readers and Plomino_Authors

Plomino_Readers contains the list of users/groups/roles allowed to read the document. By default, this item does not exist, so users defined as readers according the database ACL can read the document.

Plomino_Authors contains the list of users/groups/roles allowed to edit the document. By default, this item contains the document creator's id, appending any other author id during the document life cycle.

Those items can be easily editable using formulas:

```
# Make sure that the [purchaser] role can always edit this document
current_authors = plominoDocument.getItem('Plomino_Authors')
if '[purchaser]' not in current_authors:
    current_authors.append('[purchaser]')
plominoDocument.setItem('Plomino_Authors', current_authors)
```

onOpenDocument event

If the onOpenDocument event returns a string, it is considered as an error. The document will not be displayed, and the returned string will be displayed as a warning message.

As an example, here is one way to restrict document access to the creator of the document:

- create a field named creator (for instance). It should be of type Name and mode Computed on creation, with the following formula:

  ```
  plominoDocument.getCurrentUserId()
  ```

  This will store the user id of the user who creates the document (it might be dangerous to use the Plomino_Authors item on the document, as its value may evolve during the document life cycle).

  Add this field to the index.

- add a formula for the onOpenDocument event to make sure the user is the creator (if this formula returns a false value, opening is allowed, but if it returns a true value, e.g. a string, opening fails, and the value is displayed as an error message).

  Here's an example formula:

  ```
  member_id = plominoDocument.getCurrentUserId()
  if member_id == plominoDocument.getItem('creator'):
      return None

  roles = plominoDocument.getCurrentUserRoles()
  if "[controller]" in roles:
      return None

  return "You are not allowed to view this document."
  ```

Note: in this formula, we're checking for the [controller] custom role, instead of the PlominoManager role. While this does imply that you have to give this role to everyone who has the PlominoManager role, it allows you to distinguish between functional managers (who will only have the `[controller]` role, and technical managers (who will also have the `PlominoManager` role).

- create a search form which filters documents where the creator field matches the current user id.

CHAPTER 8

Fields reference

General parameters

id

Value Free text.

It mustn't contain special characters or spaces.

Purpose Field identifier. It is used in Plomino formulas to identify the corresponding document item.

Field type

Value

- Boolean
- Datagrid
- Date/Time
- Doclink
- File attachment
- Google chart
- Google visualization
- Name
- Number
- Rich text
- Selection list
- Text

Purpose Depending on the field type, the values entered in the field and/or the rendering of the field might be different. (See *Type-specific parameters*.)

Field mode

Value

- Editable
- Computed
- Computed for display
- Computed on creation

Purpose When editable, a field value can be entered by the user. Otherwise, its value is computed using a formula. See *Computed fields*.

Formula

Value Python code

Purpose Depending on the field mode, this formula will compute the field value (if computed), or just its default initial value (if editable). **Note:** If the field is called using `computeItem`, from some event or from some other field formula, the current field value will be overwritten even if it is an editable field.

Field read template and field edit template

Value Page template id

Purpose The custom `.pt` template to use to render the field. See *Field templates*.

Validation formula

Value Python code

Purpose The validation formula must return a string containing the error message if the validation failed, or an empty string is the validation was successful.

Add to index

Value True/False

Purpose If `True`, the field is added to the database index, allowing to perform search on its values.

Index type

Value ZCatalog index type

Purpose Allow to define how the field must be indexed. If `Default`, the field is indexed using the index type associated with its type.

Field indexing

Plomino allows you to find documents according their field values when those fields are indexed. Search can be performed using a `search form` (see related paragraph) or programmatically using the `dbsearch` method:

```
db = context.getParentDatabase()
results = db.getIndex().dbsearch({
                'Form': 'frmEmployee,
                'employee_department': 'HR'},
                sortindex='employee_name')
```

Note: `sortindex` is optionnal.

The search behaviour depends on the index types. Zope/Plone offers a standard set of indexes, and the most common ones are:

`FieldIndex`

Behaviour

- it accepts any type of values (text, dates, list, numbers, ...)
- it matches exact values,
- it allows sorting.

Example If the field value is 'Jack London', it will match if we search for 'Jack London', but not if we search for 'Jack'.

`DateIndex`

Behaviour

- it handles date values in a more efficient way than FieldIndex,
- it allows sorting,
- it allows to search using a time interval.

Examples Equality: `db.getIndex().dbsearch({'EventDate': Now()+10})` Before: `db.getIndex().dbsearch({'EventDate': {'query': Now(), 'range': 'max'}})` After: `db.getIndex().dbsearch({'EventDate': {'query': Now()-7, 'range': 'min'}})` Interval: `db.getIndex().dbsearch({'EventDate': {'query': [Now(), Now()+10], 'range': 'min:max'}})`

`ZCTextindex`

Behaviour

- it indexes text, and can match any contained word,
- it does not allow sorting,
- it allows wildcards and logical operator,
- it ignores non-meanningful words (like 'the', 'a', 'is', etc.).

Example If the field value is 'Jack London was here a long time ago', it will match if we search for:

- 'Jack London',
- 'Jack AND time',
- 'London AND NOT Paris',
- 'Lond*'.

`KeywordIndex`

Behaviour

- it indexes lists, and match their values,
- it does not allow sorting.

Example If the field value is `['Austerlitz', 'Iena', 'Waterloo']`, it will match if we search for:

- `'Austerlitz'`,

  - `['Iena', 'Austerlitz']`,
  - `{'query': ['Austerlitz', 'Agincourt'], 'operator': 'OR'}`

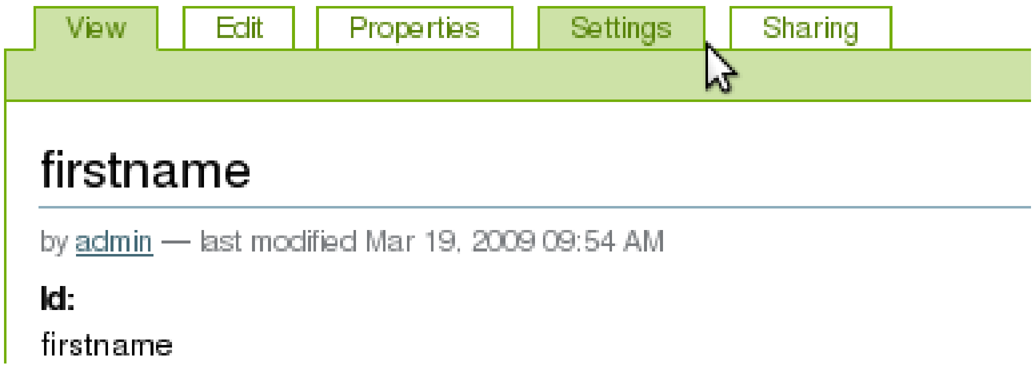
All the Plomino field types are associated to a default index type:

- Text: `FieldIndex`,
- Number: `FieldIndex`,
- Rich text: `ZCTextIndex`,
- Date/Time: `DateIndex`,
- Name: `FieldIndex`,
- Selection list: `KeywordIndex`,
- File attachment: `ZCTextIndex`,
- Doclink: `KeywordIndex`.

The default index type can be changed using the `Index type` parameter, but doing so might produce side-effects (for instance if the field was used to sort views or search results, and its type is changed to a non-sortable index, this will break sorted views).

Type-specific parameters

Type-specific parameters are available in the field's *Settings* tab:



firstname

by admin — last modified Mar 19, 2009 09:54 AM

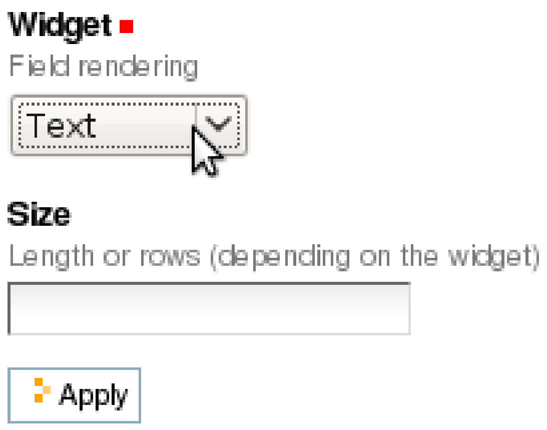
Id:

firstname

Text field

A text field allows simple text input.

Text fields can also be rendered *hidden*. In this case, input comes from the REQUEST.



Widget

Value

- `Text`
- `Long text`
- `Hidden`

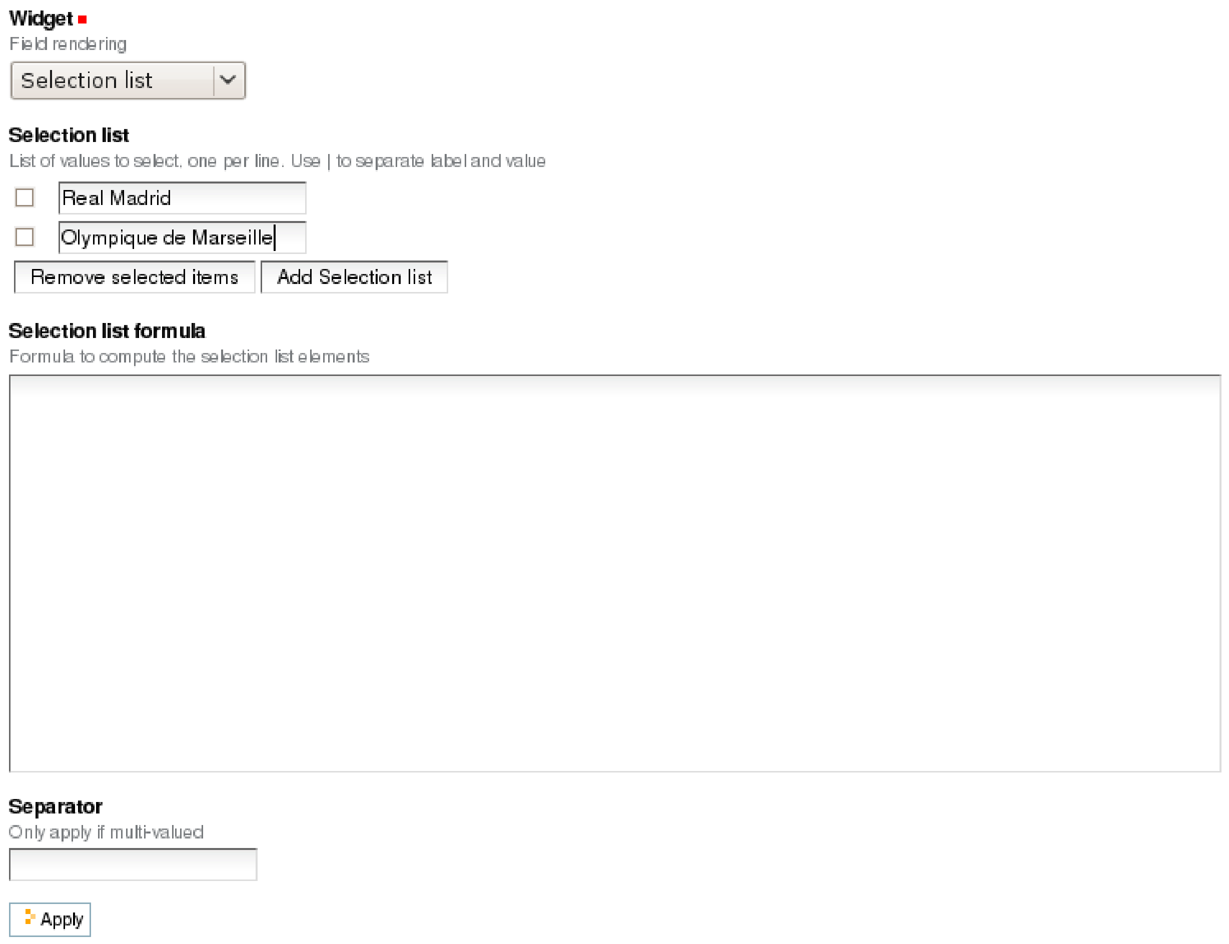
Purpose Text is rendered as a basic HTML input text field, or a hidden field.

Size

Value Integer

Purpose

- If “Text widget”: input text size.
- If “Long text”: textarea rows.


Widget
Field rendering
Selection list
Selection list
List of values to select, one per line. Use | to separate label and value
Real Madrid
Olympique de Marseille
Remove selected items
Add Selection list
Selection list formula
Formula to compute the selection list elements
Separator
Only apply if multi-valued
Apply

Boolean field

No specific parameters.

Displays as a checkbox, and stores True or False.

Selection list field

Widget

Value

- `Selection list`
- `Multi selection list`
- `Checkboxes`
- `Radio buttons`

Purpose Note: multi selection list and checkboxes are multi-valued.

Selection list

Value List of strings

Purpose The possible values selectable in the field.

Note: if a value contains a pipe (`|`), Plomino uses the string *before* the pipe as the entry label, and the string *after* as the real value.

Example: `United states of America|USA`

Selection list formula

Value Python script

Purpose The formula must return the list of values selectable in the field (using the `label|value` format if necessary).

Note: if a Selection list formula is provided, it overrides the Selection list to provide the field value list.

Separator

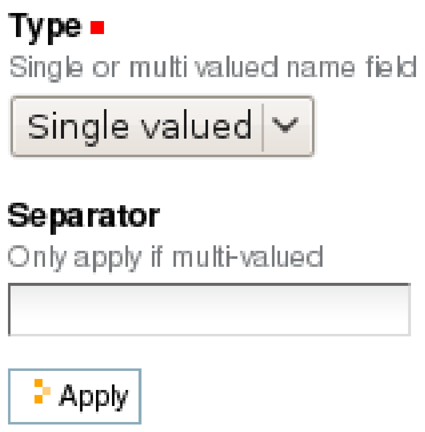
Value String

Purpose Used to separate the values in read mode for multi-valued fields and also in edit mode for radio buttons and checkboxes.

Default is blank.

Examples: `;` `-` `,`

Name field



Type

Value

- `Single valued`
- `Multi valued`

Separator

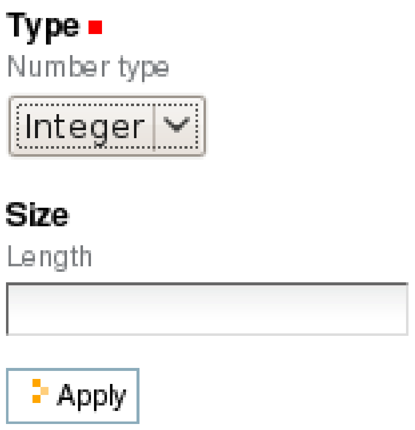
Value String

Purpose Used to separate the values in read mode.

Default is blank.

Examples: `;` `-` `,`

Number field


Type
Number type
Integer
Size
Length
Apply

Decimal numbers can be represented exactly, unlike floats. Zero is zero, not something like `5.5511151231257827e-017`.

Type

Value

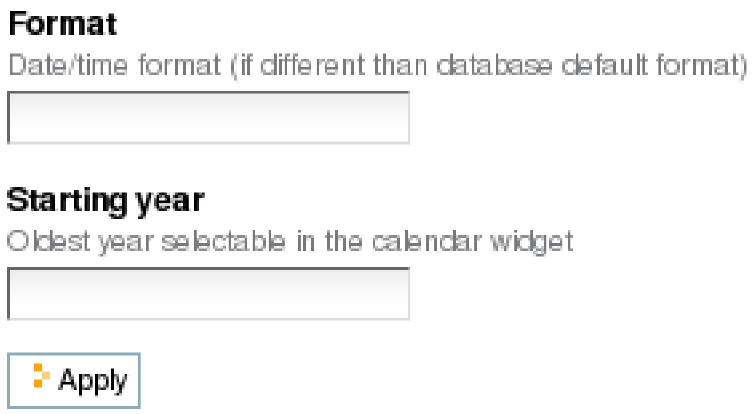
- `Integer`
- `Float`
- `Decimal`

Size

Value Integer

Purpose Length of the HTML input.

Date/Time field


Format
Date/time format (if different than database default format)
Starting year
Oldest year selectable in the calendar widget
Apply

Format

Value Python date pattern

Purpose Example: `%d/%m/%Y`

If empty, default to the Database default date format.

Starting year

Value Integer

Purpose Earliest year selectable using the date/time widget.

If empty, default to the Plone site default starting year.

Note: The calendar widget for selection of dates requires the `calendar_formfield.js` script to function. By default, this is not loaded for Anonymous users, so if you want to show a date selection widget to Anonymous, be sure to allow this script at `portal_javascripts` in the ZMI.

File attachment field

No specific parameters.

A file attachment field involves both a document item and a file. The item is named for the field and is set to a dictionary `{filename:  contenttype}` when edited through the web.

When dealing with attachment fields in formulas, both the item and the file need to be managed. For example:

```
i = 'itemname'
filename, contenttype = doc.setfile(
        myfile,
        filename='%s.csv'%i,
        overwrite=True)
doc.setItem(i, {filename: contenttype})
```

For a multi-valued field, this would be:

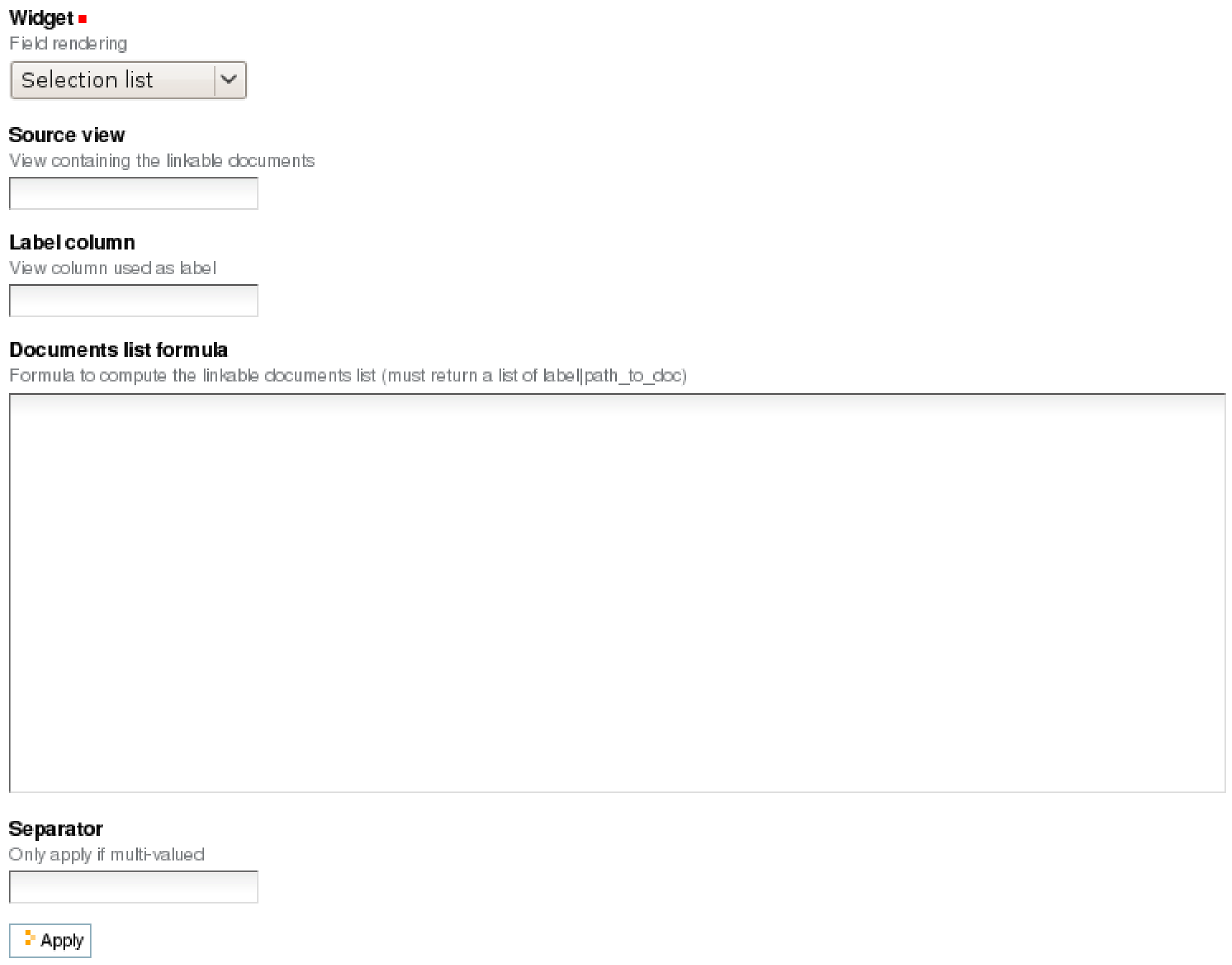
```
i = 'itemname'
filename, contenttype = doc.setfile(
        myfile,
        filename='%s.csv'%i,
        overwrite=True)
doc.setItem(i, doc.getItem(i).update({filename: contenttype}))
```

The same goes for deleting files (use the `deletefile` API).

Rich text field

No specific parameters.

Doclink field



Widget
Field rendering
Selection list

Source view
View containing the linkable documents

Label column
View column used as label

Documents list formula
Formula to compute the linkable documents list (must return a list of label|path_to_doc)

Separator
Only apply if multi-valued

Widget

Value

- `Selection list`
- `Multi-selection list`
- `Embedded view`

Purpose If *Embedded view* is selected, the view itself is displayed, with a check box on each row to allow the user to select a document.

Source view

Value The targeted view

Label column

Value The column used to provide the list labels

Purpose Only apply if Selection list or Multi selection list

Documents list formula

Value Python script

Purpose This formula must return a list of string values formatted as follows: `label|path_to_document`

Notes:

- it might a path to any kind of Plone object (even if the *main* purpose is to link to Plomino Documents),
- if a formula is provided, it overrides Source view and Label column.

Separator

Value String

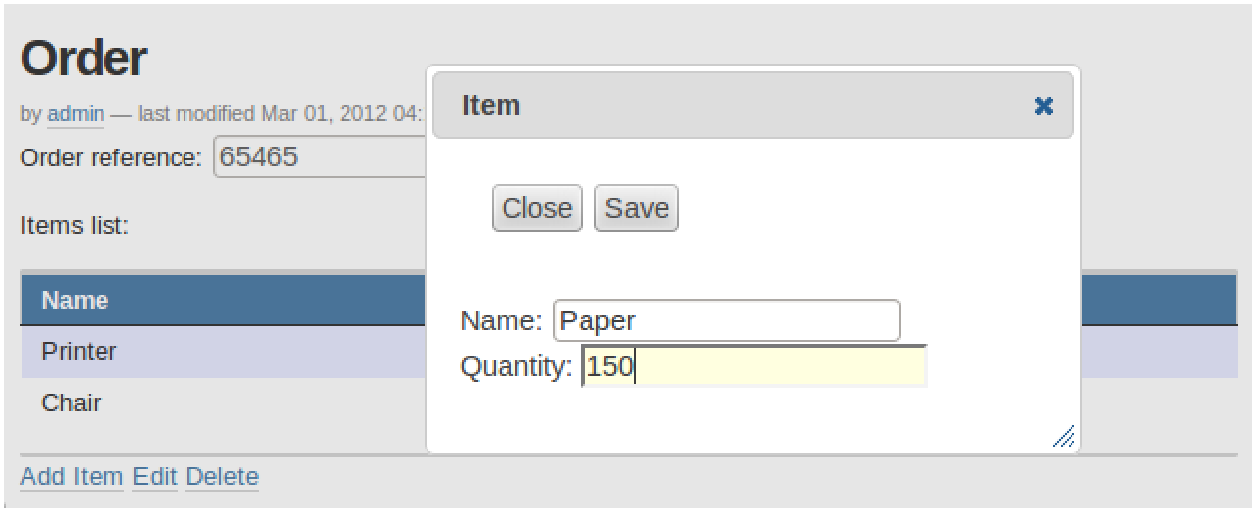
Purpose Used to separate the links in read mode.

Default is blank.

Examples: `;` `-` `,`

Datagrid field

A datagrid field allows to edit a table. Rows are edited using an associated form (displayed in a pop-up) in which fields are mapped to columns.



Associated form

Value String

Purpose Id of the form to use to add or modify row content.

Columns/fields mapping

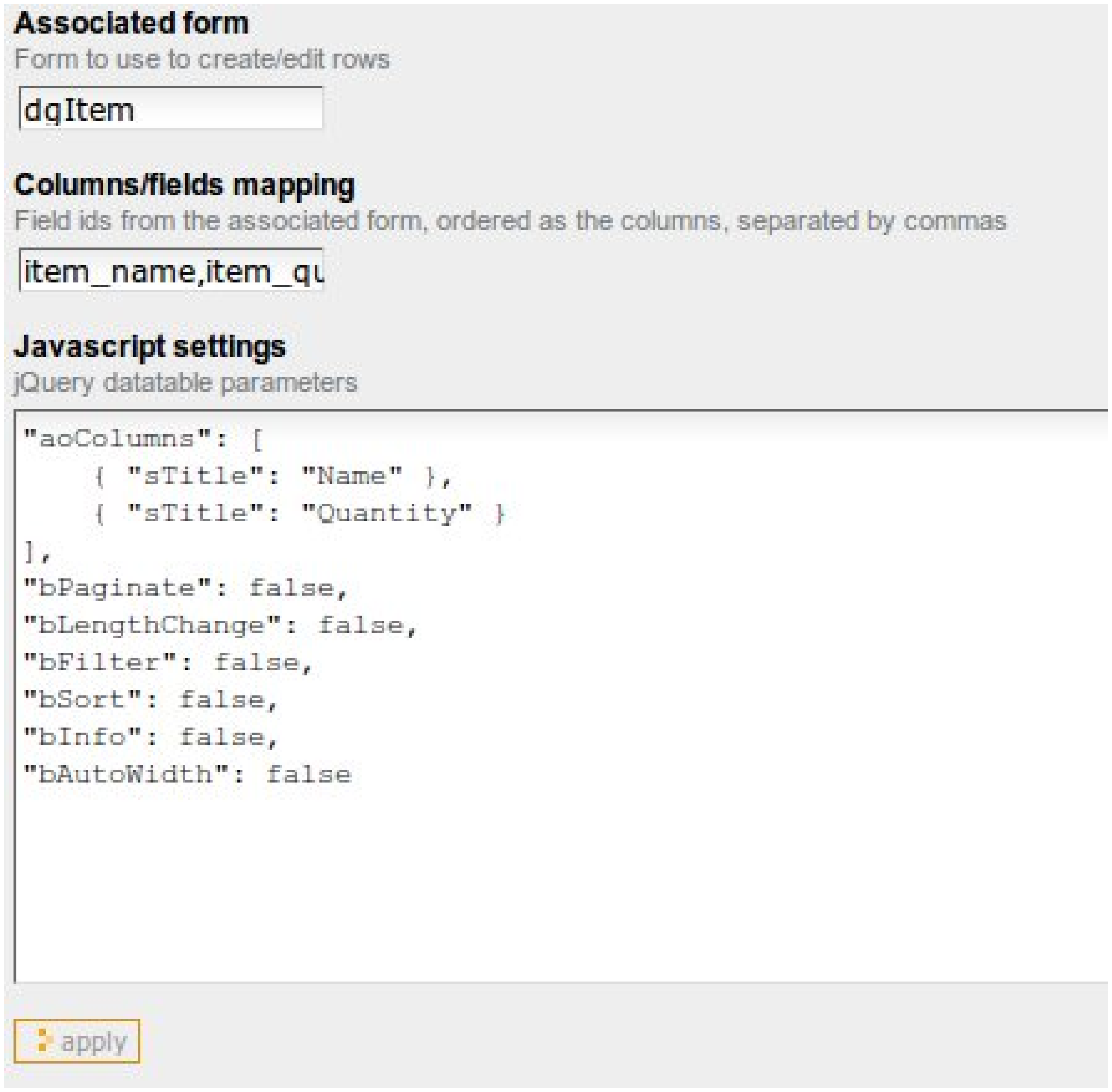
Value List separated with commas (with no space).

Purpose Field ids of the associated form sorted according the columns

Javascript settings

Value Javascript

Purpose JQuery Datatables parameters


Associated form
Form to use to create/edit rows

dgItem

Columns/fields mapping
Field ids from the associated form, ordered as the columns, separated by commas

item_name,item_qu

Javascript settings
jQuery datatable parameters

```
"aoColumns": [
    { "sTitle": "Name" },
    { "sTitle": "Quantity" }
],
"bPaginate": false,
"bLengthChange": false,
"bFilter": false,
"bSort": false,
"bInfo": false,
"bAutoWidth": false
```

Example: hide a column in a view

You can hide a column in a view by changing the *Dynamic Table Parameters* field to include something like:

```
'aoData': [{"bVisible": false}, null, null, null]
```

You would need one item in the array for each column in the table.

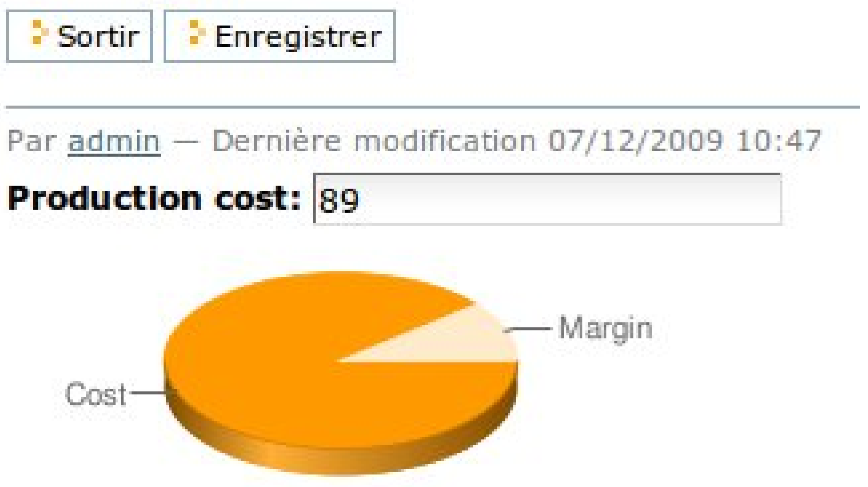
Google chart field

Allow to draw static charts (or maps, etc.).

Example

create a “Computed for display” Google chart field, and enter the following formula:

```
cost = 75
margin = 25
return {
    'chd': 't:%s,%s' % (str(cost),str(margin)),
    'chs': '250x100',
    'cht': 'p3',
    'chl': 'Cost|Margin'
    }
```



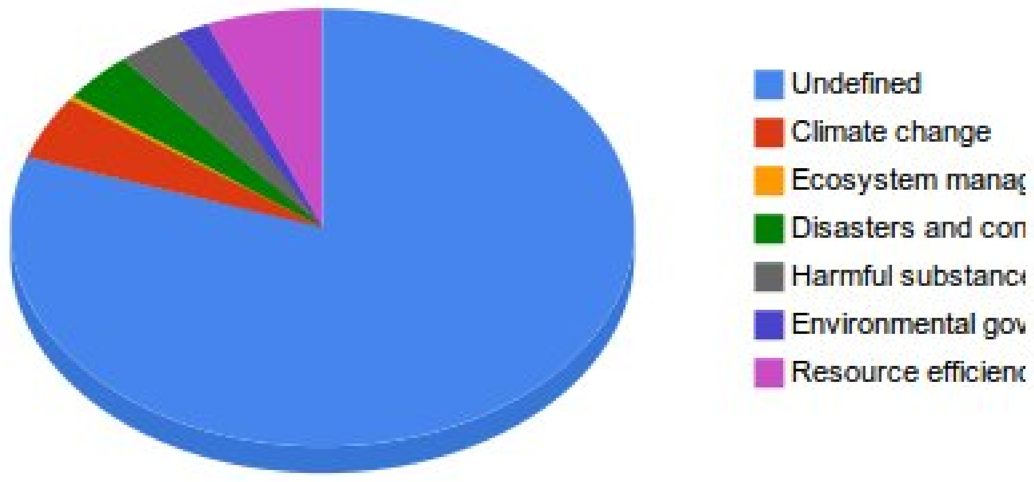
Par admin — Dernière modification 07/12/2009 10:47

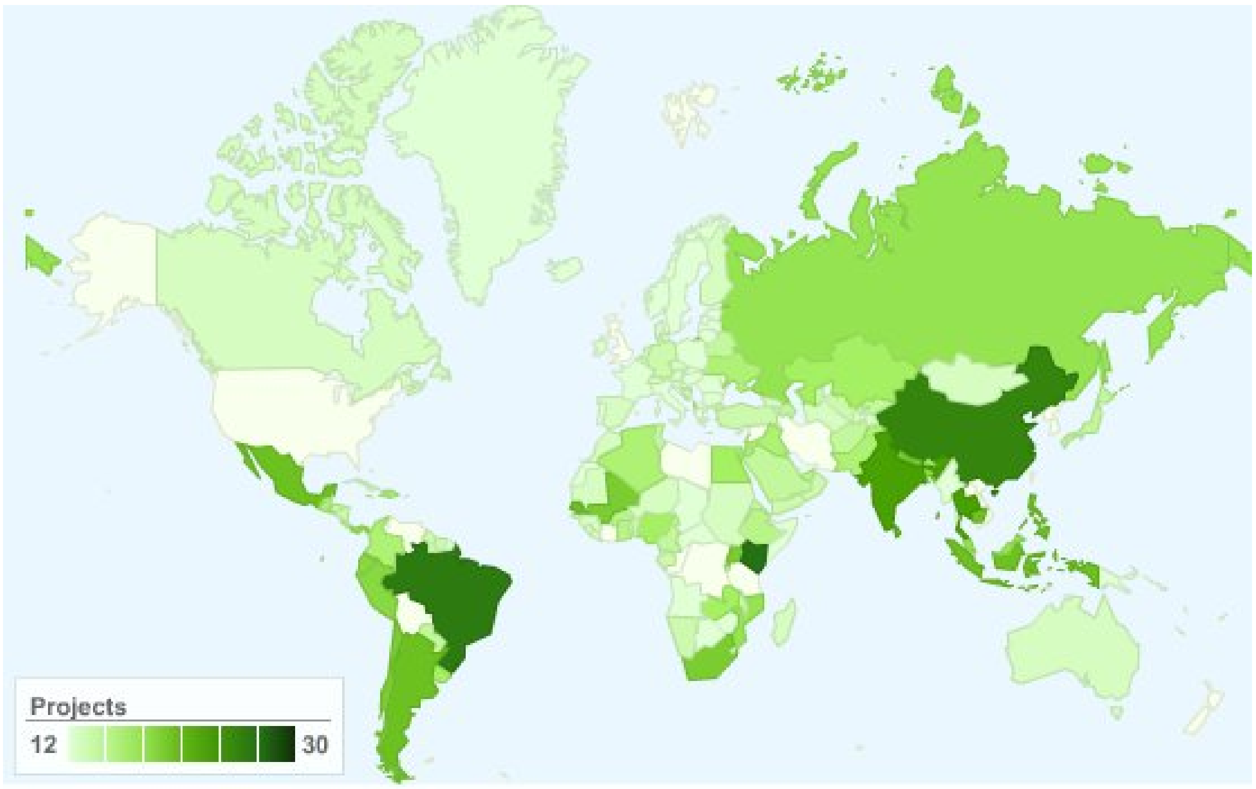
Production cost: 89

See Google chart reference.

Google visualization field

Allow to draw dynamic charts (or maps, etc.).




Projects
12
30

CHAPTER 9

Features reference

Formulas

Formulas are Python scripts. Example:

```
return plominoDocument.getItem('price') * 15
```

returns the value of the `price` item multiplied by 15.

If the `price` item doesn't exist, `getItem` will return `None`, and the formula will raise an exception. The exception is swallowed and the formula returns `None` as well. If the Plomino database is running in debug mode, the exception will be logged. If the `price` item contains a string or a truth value instead of a number, you'll also get an exception. Plomino needs to be coded defensively!

Note: a formula does not necessarily need to return a value – you may just need to make some changes in some documents (for instance if it is the formula in a Plomino action), so the return value would be irrelevant.

`plominoDocument` is a reserved name which corresponds to the current document on which the formula is evaluated.

`plominoContext` is a reserved name which corresponds to the context in which the formula is evaluated. In some cases the formula is executed on an object which is not a Plomino document (but a *view*, or a *form*, for instance).

Note: Technically, `plominoDocument` and `plominoContext` are just targeting the very same object, which is the Zope `context`. `plominoDocument`, `plominoContext`, `context` can be used identically.

Besides these names, many functions defined in `PlominoUtils` are available within the context of a formula.

Document items should be accessed using the `getItem()` method: `plominoDocument.getItem('validationDate')`.

To change an item value, use the `setItem()` method: `plominoDocument.setItem('firstname', 'Eric')`

You can access the parent Plomino database of the document (or view, or form, according the context) using the `getParentDatabase()` method.

You can also access the views and the other documents. Example:

```
db = plominoDocument.getParentDatabase()
view = db.getView('pendingPurchases')
total = 0
for doc in view.getAllDocuments():
    total = total + doc.getItem('price')
return total
```

(this example computes the total amount for the pending purchase requests).

You can check the current user rights or roles. Example:

```
db = plominoDocument.getParentDatabase()
member = db.getCurrentMember()
if db.hasUserRole(member.id, '[Expert]'):
    return True
elif db.isCurrentUserAuthor(doc):
    return True
elif 'PlominoEditor' in db.getCurrentUserRights():
    return True
else:
    return False
```

You can change the author access rights on a given document by modifying its `Plomino_Authors` item.

This item is created automatically for any document and contains the user id of the document creator. If you want your document to be editable by users other than its creator, it can contain other ids as well. Example:

```
authors = plominoDocument.getItem('Plomino_Authors')
authors.append('[Expert]')
if not 'inewton' in authors:
    authors.append('inewton')
plominoDocument.setItem('Plomino_Authors', authors)
```

As you can see in this example, you can add user ids and/or user roles.

For a better understanding of the methods available on Plomino objects, see below in this document.

Names defined in formulas

plominoContext, plominoDocument The execution context of the formula.

parent_form_ids Plomino sets the `parent_form_ids` key on the `REQUEST`, which contains a cumulative list of all the forms rendered, with the most recent form last. This allows fields on subforms to figure out what form they were rendered from.

script_id The computed id of the executing script, e.g. `field_-_frmTime_-_work_type_-_formula`. See `SCRIPT_ID_DELIMITER`.

SCRIPT_ID_DELIMITER The delimiter used in computation of script ids. To split a script into its component parts, you can do `script_id.split(SCRIPT_ID_DELIMITER)`.

Adaptive formulas

Since a formula can figure out what form and field it was called on by examining the name of its script, it can adapt to the context.

For example, to create a selection field which is initialised to select all its values by default, you can set the *default* formula as follows:

```
#Plomino import libConfig
return libConfig_getFieldValuesAsDict().keys()
```

This assumes a script library `libConfig` providing functions as follows:

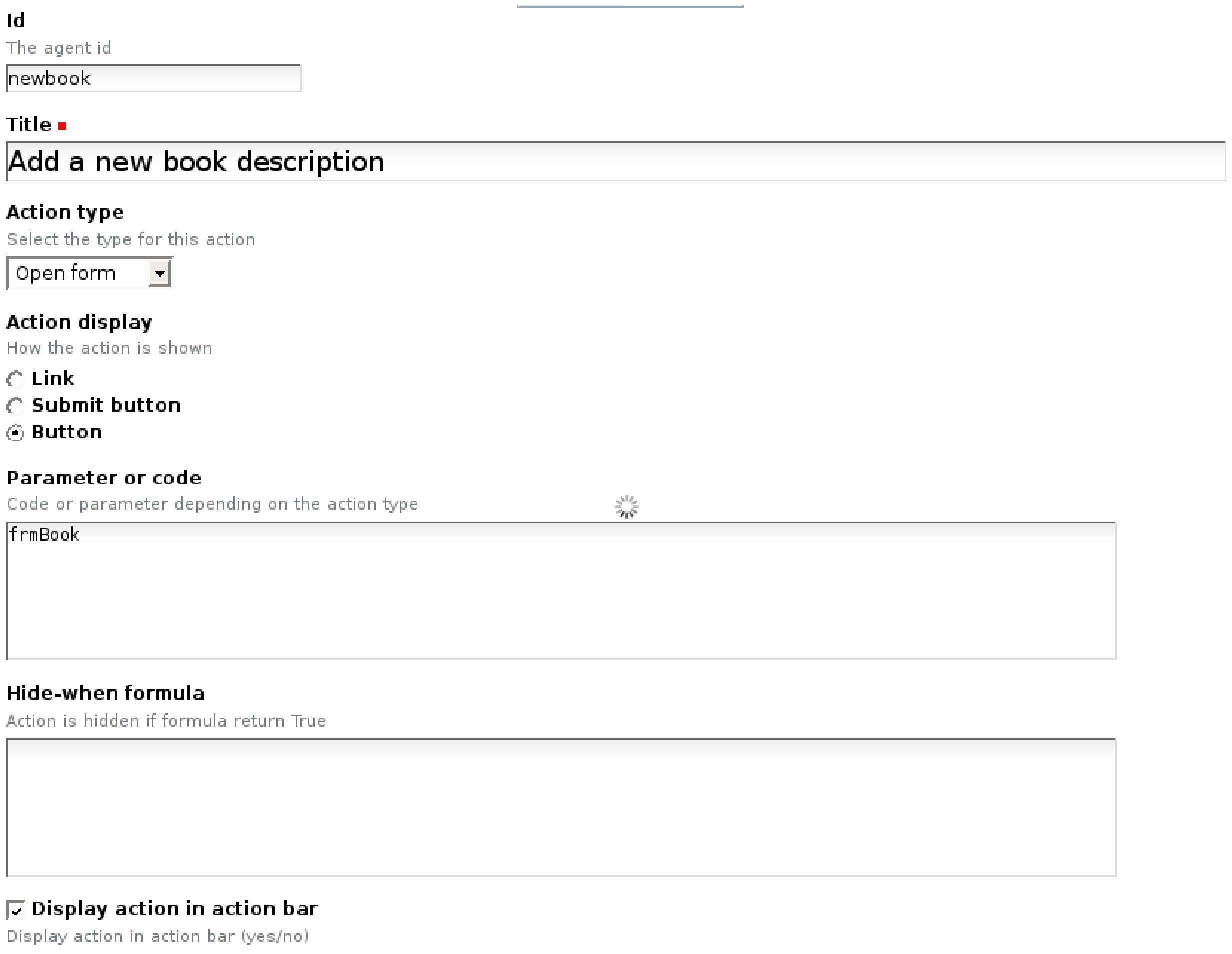
```
def libConfig_getFieldValuesAsDict():
    """ Find the config key based on the calling script name
    """
    field_id = libConfig_getFieldId(script_id)
    return libConfig_getValuesAsDict(field_id)

def libConfig_getFieldId(script_id):
    """ Parse field id from script id

    Turn this: `field-_-frmConfiguration-_-pool_construction_date-_-formula`
    into this: `pool_construction_date`
    """
    script_type, form_id, rest = script_id.split(SCRIPT_ID_DELIMITER, 2)
    field_id, formula = rest.rsplit(SCRIPT_ID_DELIMITER, 1)
    #DBG Log('field_id: %s' % field_id, 'libConfig_getFieldId')
    return field_id

def libConfig_getValuesAsDict(key):
    """ Look up a config value by name, return the value as a dictionary, splitting␣
↪each line on `separator`.

    If the selection is `['key|value', ...]`, return `{key: value, ...}`.
    """
    selection_list = libConfig_getSelectionList(key)
    d = {}
    for row in selection_list:
        (label, selection_key) = row.split(separator)
        d[selection_key] = label
    #DBG Log('values for %s: %s' % (key, `d`), 'libConfig_getValuesAsDict')
    return d

def libConfig_getSelectionList(key):
    """ Get the selection list for this field.

    This will return either:
    - the literal value or
    - the result of the selection formula, if there is one.
    """
    selection_list = []
    form_field = frmConfig.getFormField(key)
    if form_field:
        selection_list = form_field.getSettings().getSelectionList(config)
    return selection_list
```

Actions

By default, Plomino offers a few standard actions (**Exit**, **Save**, **Delete**, **Edit**, etc.) depending on the object type (document, view, form), on the current mode (read mode or edit mode), and on the user access rights.

To improve the application usability, you may need to add more actions in the forms or in the views.

To create an action, select `Plomino: action` in the *Add item* Plone menu.



Id
The agent id
newbook

Title
Add a new book description

Action type
Select the type for this action
Open form

Action display
How the action is shown
Link
Submit button
[✓ Button]

Parameter or code
Code or parameter depending on the action type
frmBook

Hide-when formula
Action is hidden if formula return True

[✓ Display action in action bar]
Display action in action bar (yes/no)

Enter an identifier and a title.

Then select the action type:

- *Open form*: open the form indicated by the `Parameter or code` parameter to create a new document.
- *Open view*: open the view indicated by the `Parameter or code` parameter.
- *Close*: go back to the database home page.
- *Save*: submit the form with its current content, and save (or create) the document.
- *Python script*: run the formula entered in `Parameter or code`, and redirect to the current object (in read mode). Examples: send a mail notification to someone, compute a field value and update the document with this new value, etc. If the formula returns a string, Plomino will assume it is a URL and use it for redirection.
- *Redirect*: similar to `Python script`, but the formula is executed *before* the document is rendered, not when the button or link is clicked. The script should return an URL which is used as a link or button target. By

including URL parameters in the returned URL, it can be used to links to a form with fields pre-filled. A better name for this action type may be *Computed link*.

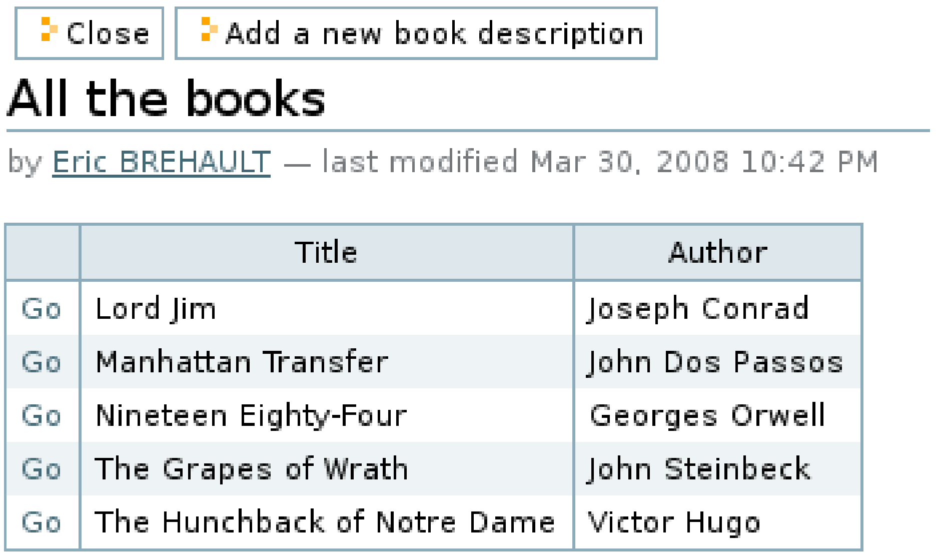
You can add a *hide-when* formula to control when the action should be visible or not.

If you select *Display action in action bar*, the action will be displayed together with the standard Plomino actions in the action bar.

But (in forms only) you can also choose to insert the action within the form layout directly.

With Tinymce, actions can be created and inserted directly in the form layout from the editor.

Example:



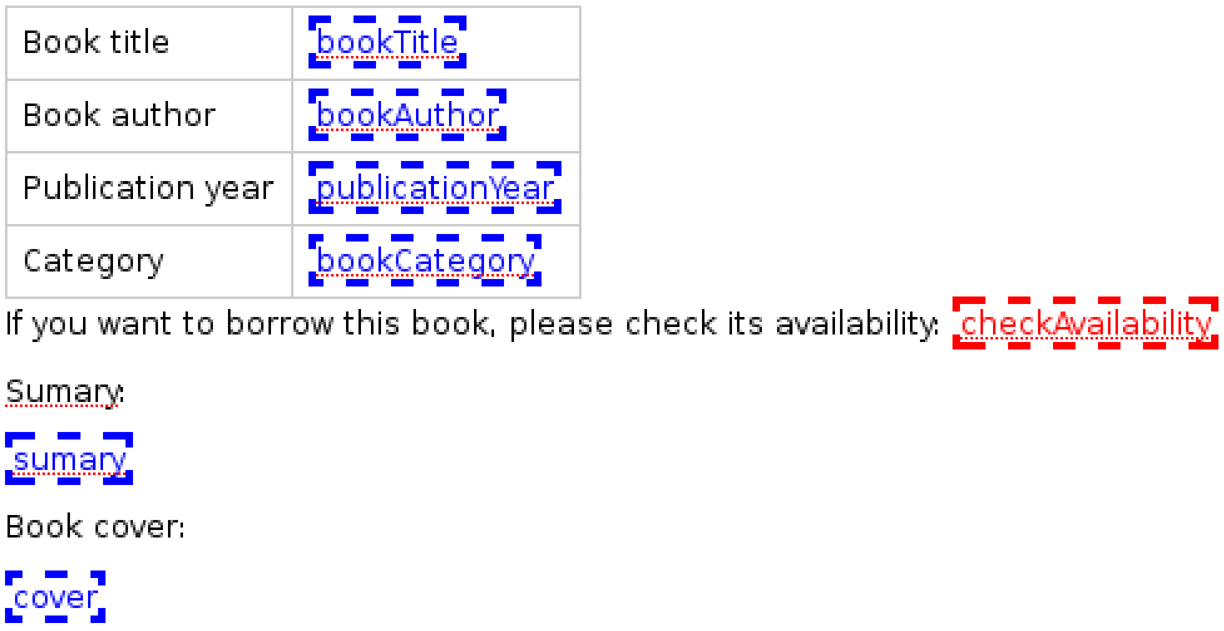
All the books

by Eric BREHAULT — last modified Mar 30, 2008 10:42 PM

	Title	Author
Go	Lord Jim	Joseph Conrad
Go	Manhattan Transfer	John Dos Passos
Go	Nineteen Eighty-Four	Georges Orwell
Go	The Grapes of Wrath	John Steinbeck
Go	The Hunchback of Notre Dame	Victor Hugo

With kupu, you must use the *Plomino action* style, and the action will be rendered according the *Action* display parameter (link, submit button, or button).

Example:



Book title	bookTitle
Book author	bookAuthor
Publication year	publicationYear
Category	bookCategory

If you want to borrow this book, please check its availability: checkAvailability

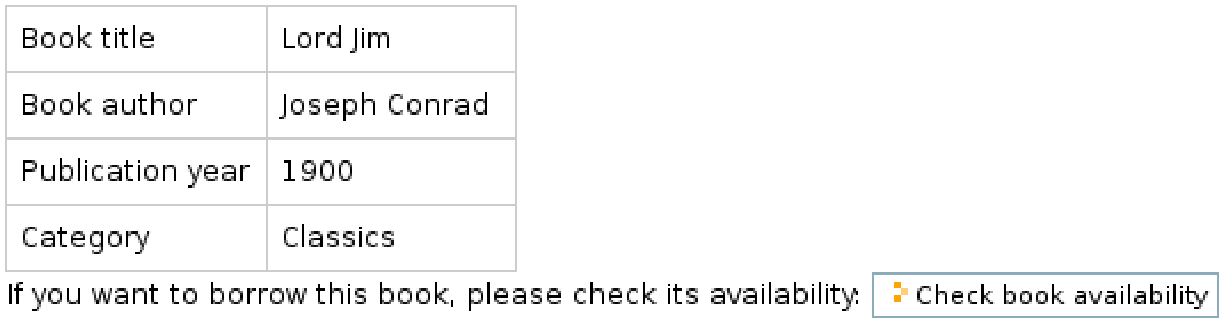
Sumary:

sumary

Book cover:

cover

Result:



Book title	Lord Jim
Book author	Joseph Conrad
Publication year	1900
Category	Classics

If you want to borrow this book, please check its availability: Check book availability

Fields

Computed fields

In a form, the fields where the user can enter data are in *edit* mode. Upon form submission, user-submitted fields are set on the document *before* computed fields are evaluated.

You might also need to use fields which can not be changed by the user. Here are the other modes offered in Plomino:

Computed the field value is computed with a formula each time the document is opened, and it is saved each time the document is saved.

Computed on creation the field value is computed only once, the first time the document is saved.

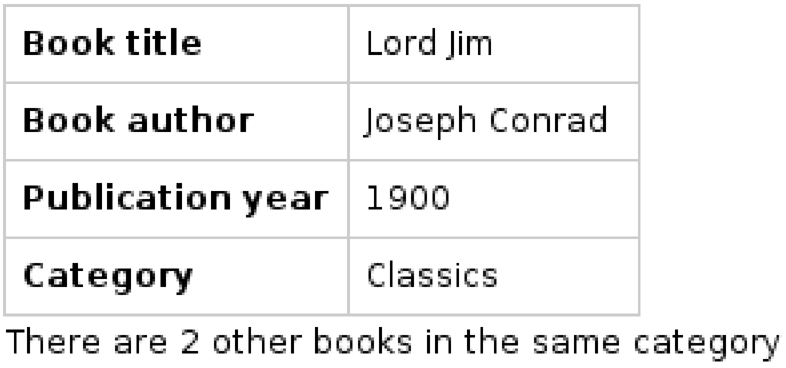
Computed on save the field value is computed and stored each time the document is saved.

Computed for display the field value is computed each time the document is opened, but it is not saved.

Example: create a *Computed for display* field with this formula:

```
category = plominoDocument.getItem('bookCategory')
if category:
    index = plominoDocument.getParentDatabase().getIndex()
    others = index.dbsearch({'bookCategory': category})
    result = "There are %s other books in the same category" % len(others)-1
    return result
return ""
```

and insert it in the `frmBook` form:



Book title	Lord Jim
Book author	Joseph Conrad
Publication year	1900
Category	Classics

There are 2 other books in the same category

A *Computed for display* field with *no* formula specified will render the item with the corresponding id, if it exists.

Note: If your computed field A depends on computed field B, the formula for A needs to compute B first. This will result in B being computed twice, so consider caching if needed.

Note: Commenting out parts of a form by editing the HTML hides that from the browser, but not from Plomino: if the commented part(s) contain computed fields, they will still be computed.

Validation

By default, Plomino validates fields according to their type (for instance, letters are not allowed in a *Number field*).

You can also add more validation criteria in the field's *Validation* tab:

- a field can be **mandatory**: if the field is empty when the document is saved, Plomino does not save the document and displays an alert to the user.
- you can enter a formula to implement a specific **validation rule** (which may depend on other field values).

For instance, in a *Purchase request* application, the maximum authorised amount is 1000 euros. You would enter the following formula in the `TotalAmount` validation field:

```
if plominoDocument.getItem('TotalAmount') >= 1000:
    return 'The total amount must be under 1000 euros'
else:
    return ''
```

If you need to compare the submitted values to the currently stored state of the document, you need to look up the stored document first:

```
db = plominoDocument.getParentDatabase()
plominoDocument_stored = db.getDocument(plominoDocument.id)
```

Note: the formula must return an empty string if validation succeeds or an error message if it does not.

Doclinks

A Doclink field allows a Plomino document to reference another document.

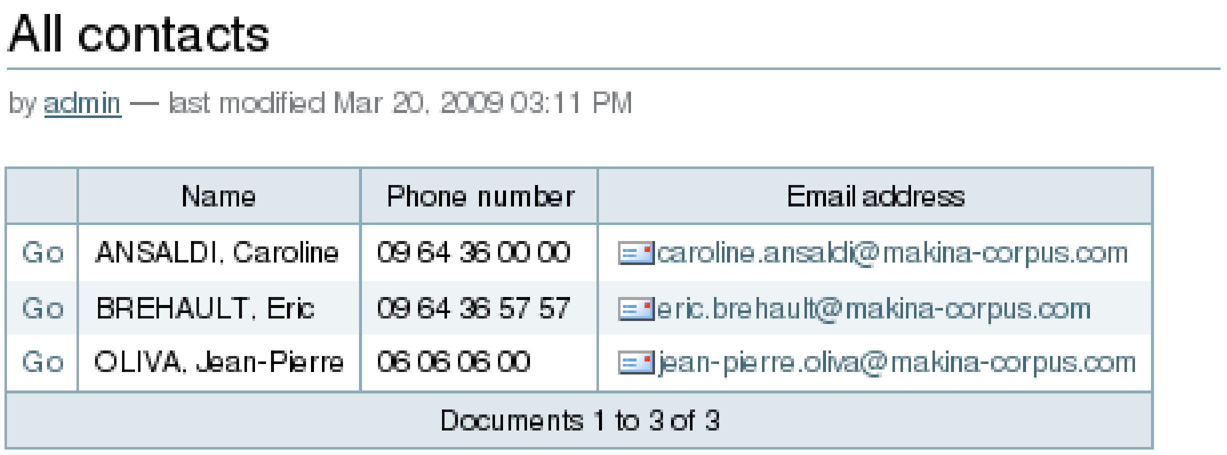
The doclink stores the path to the targeted document, and it is displayed as a link.

You may use a *view* as the source of possible targeted documents.

Example

We have a Contact form allowing users to enter contact information about company employees.

The view `allcontacts` returns all the contact documents:

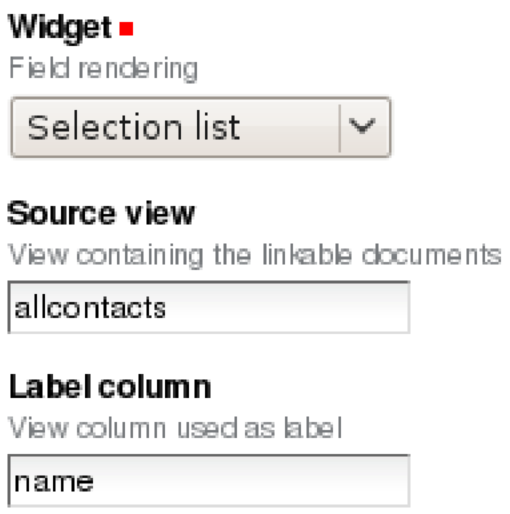


All contacts

by admin — last modified Mar 20, 2009 03:11 PM

	Name	Phone number	Email address
Go	ANSALDI, Caroline	09 64 36 00 00	caroline.ansaldi@makina-corpus.com
Go	BREHAULT, Eric	09 64 36 57 57	eric.brehault@makina-corpus.com
Go	OLIVA, Jean-Pierre	06 06 06 00	jean-pierre.oliva@makina-corpus.com
Documents 1 to 3 of 3			

We add a doclink field, named `manager`, in the Contact form to enter the corresponding manager of each employee.

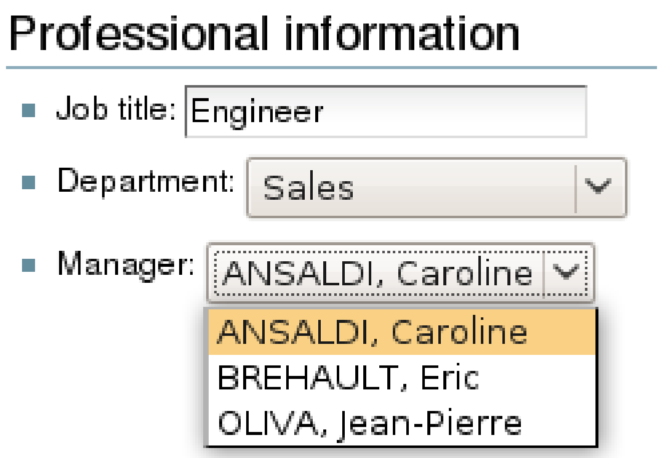


Widget
Field rendering
Selection list

Source view
View containing the linkable documents
allcontacts

Label column
View column used as label
name

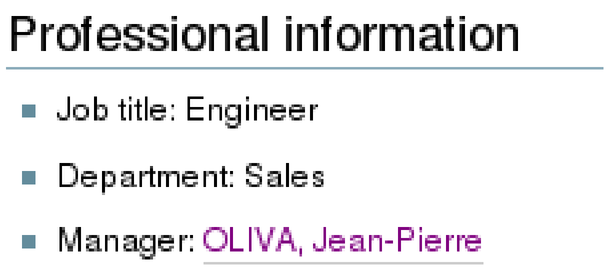
It uses the `allcontacts` view as its document list source, and displays the `name` column value as label:



Professional information

- Job title: Engineer
- Department: Sales
- Manager: ANSALDI, Caroline

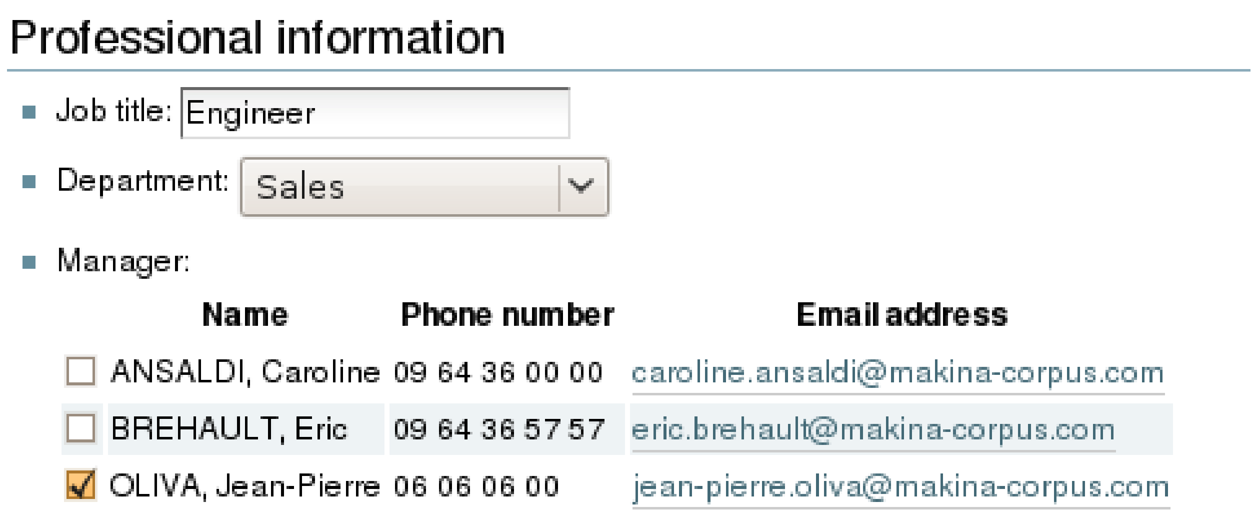
In read mode, the field displays a link to the corresponding document:



Professional information

- Job title: Engineer
- Department: Sales
- Manager: OLIVA, Jean-Pierre

If you choose the `Embedded view` widget, the field displays the view itself (including all columns), with a checkbox to select documents:


Professional information

- Job title: Engineer
- Department: Sales
- Manager:

	Name	Phone number	Email address
☐	ANSALDI, Caroline	09 64 36 00 00	caroline.ansaldi@makina-corpus.com
☐	BREHAULT, Eric	09 64 36 57 57	eric.brehault@makina-corpus.com
✓	OLIVA, Jean-Pierre	06 06 06 00	jean-pierre.oliva@makina-corpus.com

Instead of using a view, you can compute the document list using a formula (which will override the Embedded view widget), so you can filter the documents you want to list, you can retrieve documents from another database, or even list Plone objects which are not Plomino documents. For example:

```
contactsdb = plominoDocument.restrictedTraverse("/Plone/demo/contacts")
view = contactsdb.getView('allcontacts')
return [d.lastname+"|"+d.getPath() for d in view.getAllDocuments()]
```

Note: in this example, we use the `getAllDocuments` method to get the documents list, this method returns Catalog brains.

To improve performance, the `lastname` field has been added to the index, so there is no need to wake up the objects (using `getObject`), and we use the `getPath` method to get the path of the real object.

Field templates

You can create a custom template to render a field in a different way than the regular field widgets.

The field template must be added in the Resources folder in the ZMI (go to *Design* tab / *Others* / *Resources folder*) as a Page Template.

To be applied, the template id must be then entered in the *Field read template* or in the *Field edit template*.

The template code can be copied from the Plomino products sources (*CMFPlomino/skins/cmfplomino_templates/**FieldEdit.pt* or ***FieldRead.pt*).

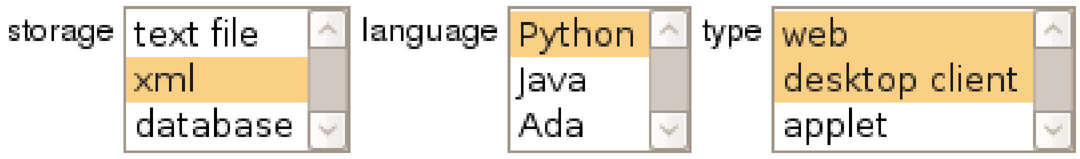
Here is an example showing a multi-categorized tag field:

Edit template:

```
<span tal:define="
    field options/field;
    db options/field/getParentDatabase;
    categories python:
        [doc.getObject() for doc in db.getView('tags').getAllDocuments()]
    ">
<table><tr>
<tal:loop repeat="cat categories">
```

```
    <td valign="top" tal:define="
        c cat/tagCategory;
        tags cat/tagList
        ">
    <span tal:content="c">category</span>
    <select tal:attributes="name options/fieldname"
        multiple="true"
        lines="4">
    <tal:block repeat="v python:
        [t+'|'+c+':'+t for t in tags.split(',')]
        ">
        <tal:block define="
            current options/fieldvalue;
            l python:v.split('|')
            ">
            <option tal:attributes="
                value python:l[1];
                selected python:test(current and l[1] in current,1,0)
                "
                tal:content="python:l[0]">value</option>
        </tal:block>
    </tal:block>
    </select>
    </td>
</tal:loop>
</tr>
</table>
</span>
```

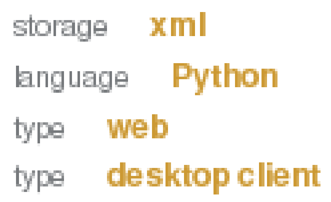
Result:



Read template:

```
<tal:block tal:repeat="v options/selection">
    <tal:block define="
        current options/fieldvalue;
        l python:v.split('|')
        ">
        <tal:block condition="
            python:test(current and l[1] in current,1,0)">
            <tal:block define="
                v python:l[0];
                cat python:v.split(':')[0];
                t python:v.split(':')[1]
                ">
        <br/><span class="discreet" tal:content="cat">category</span>
        <span class="callout" tal:content="t">category</span>
            </tal:block>
        </tal:block>
    </tal:block>
</tal:block>
```

Result:



storage	xml
language	Python
type	web
type	desktop client

Filling fields from the REQUEST

Editable fields which are not part of the layout take their value from the `REQUEST`.

So, for example, if you want to pass a parameter to another form:

- in the origin document, put the parameter(s) in the link to the target form, e.g. by adding `?param1=value&param2=value` to the URL. This will cause the parameter to be part of the `GET` request which retrieves the target form.
- in the target form, create an editable field with the same id as the parameter key (e.g. `param1` and `param2` above), but do not insert it in the form layout. The field will get its value from the `REQUEST`. - then you can create *Computed on save* (or on display, or whatever) fields which use the value of this field.

Field labels

Form layouts may contain field labels. See **'field labels'_** below.

HTML attributes injection

The HTML attributes formula parameter allows to enter a formula in charge of returning a string that will be injected in the field tag element.

A typical example could be:

```
'placeholder="Enter the book title here"'
```

But it also be a good way to enable Mockup pattern-based widgets:

```
'class="pat-select2" data-pat-select2="width:20em"'
```

or basically any HTML attributes a Javascript library or a CSS grid might expect.

Note: the attributes are inserted just before 'id="field_id"' if it exists.

Forms

Document id and title formulas

Document title formula Compute the document title

Compute document title on view Execute the document title formula whenever the document is rendered

Store dynamically computed title Store the computed title (if different from the stored value) every time the document is rendered. (Watch out, this can become a hotspot if it causes many writes.)

Document id formula Compute the document id at creation. (Undergoes normalization.)

Field labels

A field label corresponds to a field. To create a label, add text with the format `fieldid: Label` or just `fieldid` to the layout, select this text, and select the *Plomino Label* style from the TinyMCE styles dropdown.

The `fieldid` has to correspond to a field in the layout.

If no label is specified (i.e. `fieldid`), the field title is used as the label.

In *edit* mode, labels for single-input fields are rendered as an HTML `<label for='FIELDID'>LABEL</label>` element.

In *read* mode, labels for single-input fields are rendered as an HTML `<span class='label' title='Label for FIELDID'>LABEL</span>` element.

In *edit* mode, labels for composite fields such as checkboxes, radio buttons, and picklists are rendered as a `<fieldset><legend>LABEL</legend>...</fieldset>` structure, wrapping the target field.

In *read* mode, labels for composite fields are rendered as a `<div class='fieldset'><span class='legend' title='Legend for FIELDID'>LABEL</span>...</div>` structure.

Note that `label` elements are rendered in-place (which may be anywhere in the layout), while `fieldset` elements are rendered around the target field.

Events

In a Plomino form, you can use the following events:

`onOpenDocument` executed before document is opened (in both read mode and edit mode)

If the formula for this event returns a false value, opening is allowed; but if it returns a true value, e.g. a string, opening fails, and the value is displayed as an error message.

`beforeSaveDocument` executed before submitted values are stored into the document. Submitted values can be found in `context.REQUEST`.

`onSaveDocument` executed after editable and computed items have been stored, and before document is re-indexed.

`onDeleteDocument` executed before document is deleted

`onCreateDocument` executed before the document is saved for the first time (`onSaveDocument` will also be executed, but after `onCreateDocument`)

`beforeCreateDocument` executed before a blank form is opened.

In the *Events* tab, you can enter the formulas for each event you need.

Example: enter the following formula for the `onSaveDocument` event:

```
date = DateToString(DateTime())
db = plominoDocument.getParentDatabase()
user_name = userFullname(db, db.getCurrentMember())
plominoDocument.setItem(
    'history',
    plominoDocument.getItem('history') +
    "This document has been modified by "+user_name+" on "+date)
```

It will update the `history` item which logs all the modifications, authors and dates.

Hide-when formulas

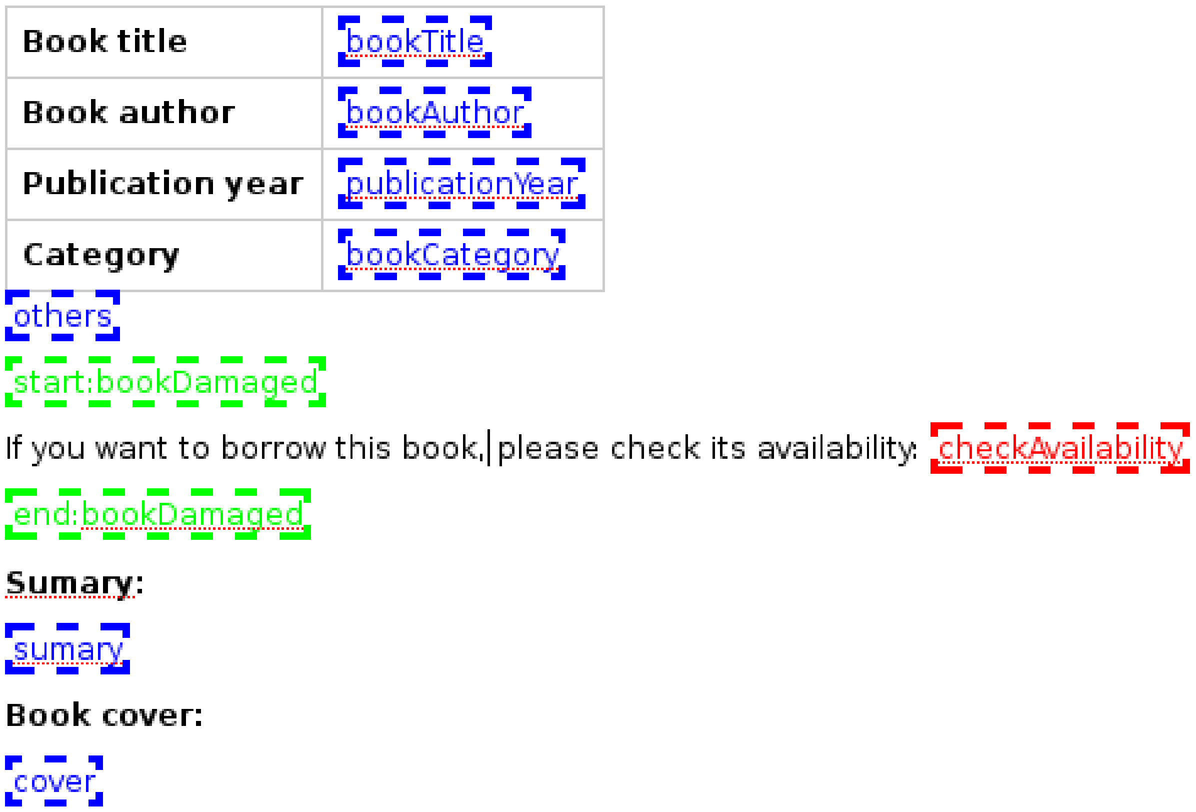
In a form, it might be useful to hide or display some sections according different criteria (an item value, the current date, the current user's access rights, etc.).

To do so, you must use Hide-when formulas.

Select *Plomino: hide when* in the *Add item* Plone menu.

Enter an identifier, a title, and a formula. Example: `plominoDocument.bookState == 'Damaged'`

Then, modify the form layout to insert the hide-when formula in the form layout. Enter the following: `start:hide-when-identifier` at the beginning of the area to hide. And the following at the end: `end:hide-when-identifier` And apply the Plomino *Hide-when* formula style to those 2 bounds:



Book title	bookTitle
Book author	bookAuthor
Publication year	publicationYear
Category	bookCategory

others

start:bookDamaged

If you want to borrow this book, please check its availability: checkAvailability

end:bookDamaged

Sumary:

sumary

Book cover:

cover

If the *hide-when* formula returns `True`, the enclosed area will be hidden. If it returns `False`, the area is displayed (in our example: if the book is damaged, it cannot be borrowed, so we hide the action to check the book availability).

Hide-when formulas can be inserted directly in the form layout using TinyMCE.

Sub-forms

An application can contain several forms.

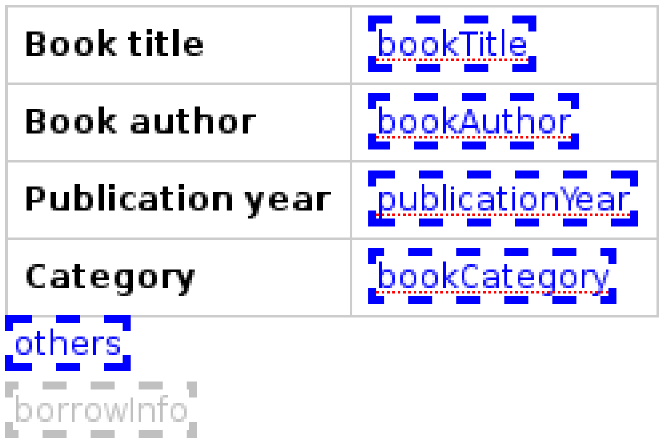
In the Book library example, we could add a CD form and a Video form. Those two forms would probably have several similar fields (availability, last borrower, return date, etc.).

To avoid having to build (and maintain) the same things several times, you can use sub-forms.

The sub-form principle is to insert a form within another form.

In our example, we create a `borrowInfo` form containing the borrower name, the return date, and the availability, and we insert it as a sub-form in `frmBook`, `frmCD` and `frmVideo`.

The form is inserted using the Plomino *Subform* style in Kupu:



Book title	bookTitle
Book author	bookAuthor
Publication year	publicationYear
Category	bookCategory

Sub-forms can be inserted directly in the form layout using TinyMCE.

Note: as you probably do not want `borrowInfo` to be displayed in the database home page, you have to check *Hide in menu* in the form *Parameters* tab.

Note: Some fields type are computed independently of rendering, namely `COMPUTED`, `COMPUTEDONSAVE` and `CREATION`. In the case of sub-forms, if multiple sub-forms have fields with the same id as the including form, or other included forms, those fields will be found multiple times. Plomino handles this case by picking the first occurrence of the field, and logging the ambiguity (at the `WARNING` log level).

Search formula

When you create a search form, Plomino uses the form fields to do a default ZCatalog search among the documents of the view associated with the search page.

If needed, you can create a specific search formula in the form *Parameters* tab.

This formula is used to filter the result set of the default query, and must return `True` or `False` for each document in the result set.

You can access the values submitted by the search form on the `REQUEST` object: `plominoContext.REQUEST.get('myfield')`.

Example:

```
period = plominoContext.REQUEST.get('period')
if period == 'Ancien regime':
    return plominoDocument.year
if period == 'Empire':
    return plominoDocument.year >= 1804 and plominoDocument.year
```

Note: Search formulas can be a lot slower than regular ZCatalog searches, you must use them carefully.

Search event

If you do not want the default filters of a search page (the view, the query, and the formula), you can define an `onSearch` event on the form *Events* tab. The formula of this event should return the required list of documents.

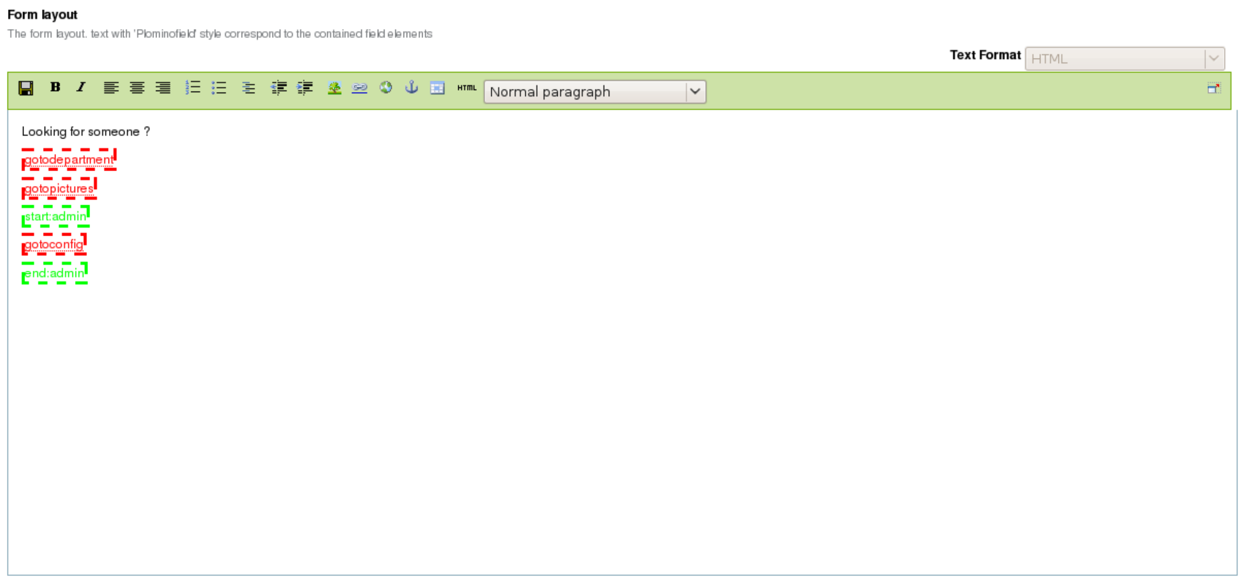
You can access the values submitted by the search form on the `REQUEST` object: `plominoContext.REQUEST.get('myfield')`.

Page

Like a *Search* form, a *Page* form cannot be used to save documents through the web, since *Page* forms do not display any action bar. (Formulas could however still call `save` on a document using a Page form.)

Like any form, it can contain computed fields, actions (inserted in the form layout), and hide-when formulas, so it is a good way to build navigation pages, custom menus, or information pages (like reports, etc.).

Example:



Form layout

The form layout. text with 'Plominofield' style correspond to the contained field elements

Text Format: HTML

Normal paragraph

Looking for someone ?

gotodepartment

gotopictures

start:admin

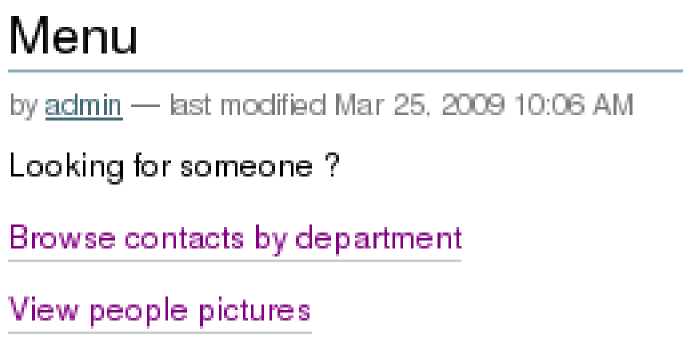
gotoconfig

end:admin

Here we create a page with 3 actions to access 3 different views, but the last one is enclosed in a *Hide-when* formula so it will not be displayed if the current user does not satisfy a given criterium. In the example, we test if the user has the `[dbadmin]` role:

```
"[dbadmin]" not in plominoContext.getCurrentUserRoles()
```

Result if you are not `[dbadmin]`:



Menu

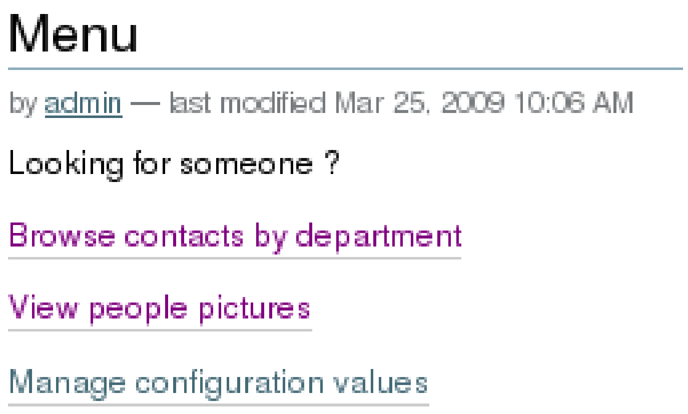
by admin — last modified Mar 25, 2009 10:06 AM

Looking for someone ?

Browse contacts by department

View people pictures

Result if you are `[dbadmin]`:



Menu

by admin — last modified Mar 25, 2009 10:06 AM

Looking for someone ?

Browse contacts by department

View people pictures

Manage configuration values

Open-with form

The form used to render a document is determined by a number of mechanisms:

- By default, Plomino document is displayed using the form corresponding to its `Form` item value (which contains the id of the form last used to save the document).
- If the view from where the document is opened defines a `Form` formula, the resulting form will be used instead.
- And to force the usage of a given form, the form id can be passed in the request using the `openwithform` parameter.

Example:

http://localhost:8080/test/testdb/58862f161ea71732944d37e0a0489cfc?openwithform=frmtest

Accordions and lazy loading

In Plomino it is possible to *accordion* some parts of the page. This means that the content of the accordioned part will not be visible unless you click on the headline to open the accordion.

It is also possible to avoid loading the content of the accordion until such time as the accordion is opened. This is particularly useful if the content it very big, or if there are many accordions on a page and the reader is interested in only a few of them.

To turn part of a page into an accordion, use this structure (the header level can be from `h2` to `h6`):

```
<h5 class="plomino-accordion-header"><a href="TARGETURL">Header</a></h5>
<div>Content</div>
```

If the class is `plomino-accordion-header`, the content of the page referenced by `TARGETURL` will be substituted for the subsequent div.

Note: Plomino does not currently offer UI support for this functionality. To use it, you have to generate the desired content via Python, or enter it literally into the form layout.

Caching

To improve performances, it might be useful to cache some fragments of a form so they are not re-computed every time.

Cached fragments are set in the layout the same way as hide-when formulas, with `start:cache-identifier` and `end:cache-identifier` markers. The associated formula is supposed to return a cache key.

When the form is rendered the first time, the resulting HTML contained into the delimited area will be stored in cache and associated with the cache key. Every time the form is rendered, if the cache key returned by the formula matched an existing cache key, the cached HTML is returned.

Consequently, if you use a formula returning always the same value, e.g.:

```
"financial-report"
```

the same cached fragment will be served to all the users in all the cases.

If you use a formula which depends on the current user, e.g.:

```
"personal-report-" + context.getCurrentUserId()
```

then there will be a different cached fragment for each user (so if the same user displays the form twice, she will received the cached content the second time, but other users would not get that cached fragment, they would get their own cache).

The formula might depend on the date:

```
"today-report-" + DateToString(Now(), "%Y-%m-%d")
```

or anything (the document id, any specific item value, etc.).

If the cache key is `None`, caching is not applied, so for instance:

```
if context.isEditMode():
    return None
else:
    return "something-read"
```

would show the cached content in read mode, but would always regenerate the content in edit mode.

Specific CSS or JS

If a form needs some specific CSS or JS, they can be mentioned in the form edit page in the 2 respective textarea fields (one URL per line).

Those URLs can target:

- a file provided in the theme but not enabled in `portal_javascript`,
- a file contained in the Plomino database `resources` folder,
- an external file (most likely a CDN URL).

Views

Form formula

You may need to read or edit documents using different forms.

For instance, a person who wants to borrow a book wants different information (book description, category, publication year, etc.) than the librarian (who may want last borrower, return date, availability, etc.).

As explained previously, we can manage this issue using *hide-when* formulas, *action*'s and *sub-form*'s.

But if the functional differences are too great, or if the layout is totally different, those strategies will probably produce too much complexity.

In such a case, it is better to create a totally different form (named `frmBorrowManagement` for instance).

However, by default the document opens with the form used the last time it was saved.

To open the document with a different form, you need to create a specific view for borrowing management and use the `Form` formula parameter.

This formula will compute the name of the form to use when the documents are opened from the view.

If you enter `frmBorrowManagement` in Form formula, all the documents opened from this view will be displayed using the `frmBorrowManagement` form.

View template

If you need a specific layout for a view, you can create a ZPT page which can be used instead the default template.

This way, you can build calendar views, Gantt views, produce charts, etc.

To do so, add your Page Template in the resources folder, and enter its name in *View Template* in the view *Parameters* tab.

A good approach is to copy the ZPT code from `CMFPlomino/skins/CMFPlomino/OpenView.pt` (in the Plomino sources) and add your modifications.

Note: good knowledge of ZPT is required.

Export CSV

All the views can be exported as CSV. The export contains the value of each column.

Go to the database *Design* tab, expand the *Views* section and click the green arrow icon next to the view you want to export.

You can build views specifically for export purposes, you just need to create the columns according the values you want to get in CSV (note: if you do not want this view to be offered on the database home page, check *Hide in menu* in the view *Parameters* tab).

Database

Design import/export

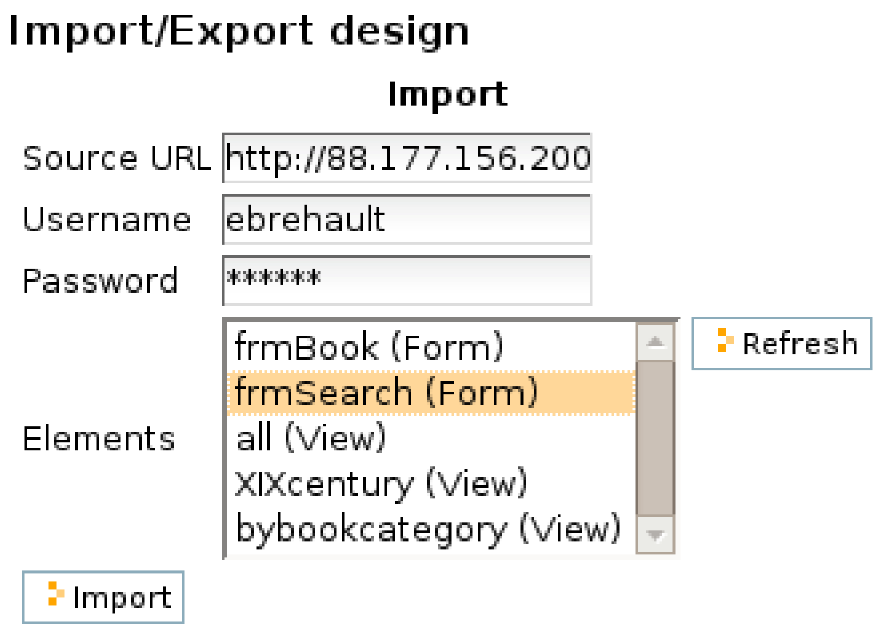
You can export or import Plomino database design elements from one Zope instance to another.

This may be useful if you want to deploy a new application from a development server to a production server, or if you want to release a modification or a correction on an application already in production.

To import design elements, go to the database *Design* tab, and in the *Import/Export Design* section, fill in the following parameters:

- the URL of the Plomino database which contains the elements you want to import in the current database;
- user id and password corresponding to a user account on the remote instance. This account must be Plomino-Manager on the remote Plomino database.

Then click on refresh: Plomino will load the list of all the available elements in the remote database.



Import/Export design

Import

Source URL	http://88.177.156.200
Username	ebrehault
Password	******
Elements	frmBook (Form) frmSearch (Form) all (View) XIXcentury (View) bybookcategory (View)

You can then choose the elements you want and click on *Import* to import them into the local database.

In some cases (depending on firewalls, proxies, etc.), it is easier to export from the local database to the remote one.

The principle is the same, you just need to use the *Export* section.

Design export/import as genericsetup resources

Databases can be made available as genericsetup resources. The main purpose for this is to allow them to be used as templates when creating a new database, and so they are referred to as **template databases**

Plomino defines *Export Plomino templates* and *Import Plomino templates* steps for *genericsetup*.

The *export* step will search for all Plomino databases contained in the portal. If the database has the `IsDatabaseTemplate` checkbox ticked, its design will be included in the export. There is no difference between a database marked as template and any other database, it merely makes the database available as a template.

The databases are written to folders `plomino/<dbid>/` in the exported resource archive (`.tar.gz`), where `<dbid>` is the database id.

Exported database resources can be included e.g. in a Plone skin product.

When adding a new Plomino database to a Plone instance with such a skin installed, templated databases included as resources are offered as starting point for the new database.

This is useful in a hosted environment, to make preconfigured Plomino databases available as two- or three-click installs (add database, choose template, go) as starting point for a user. Once imported, the template forms are part of the user's database, and edited along with the user's own forms.

For this use, the hoster would have a source Plone instance containing all the databases that they want to make available together, for example via a specific skin. The source Plone serves to define a group of database templates. Mark all these databases as templates, and export them as a genericsetup resource archive.

Step by step procedure:

- For each database you want to provide as template, go to its *Parameters* page, and enable *Use as a template*.
- Go to your Plone portal ZMI / portal_setup / Export page,
- select the *Export Plomino templates* step, and click *Export selected steps*. This produces a `.tar.gz` file.
- Go to the Plone site where you want to provide those db templates,
- go to its ZMI / portal_setup / Import page, and at the bottom, import the previously downloaded `.tar.gz` resource archive file.
- Now create a new Plomino database in your site. The default welcome page will provide a list of the available templates, so you can pick one and get its design immediately imported in your database.
- the template selection is also available in the *Database Design* tab.

Refresh a database

After copy/paste of views or forms, or deletion of fields, a Plomino database may be out of date.

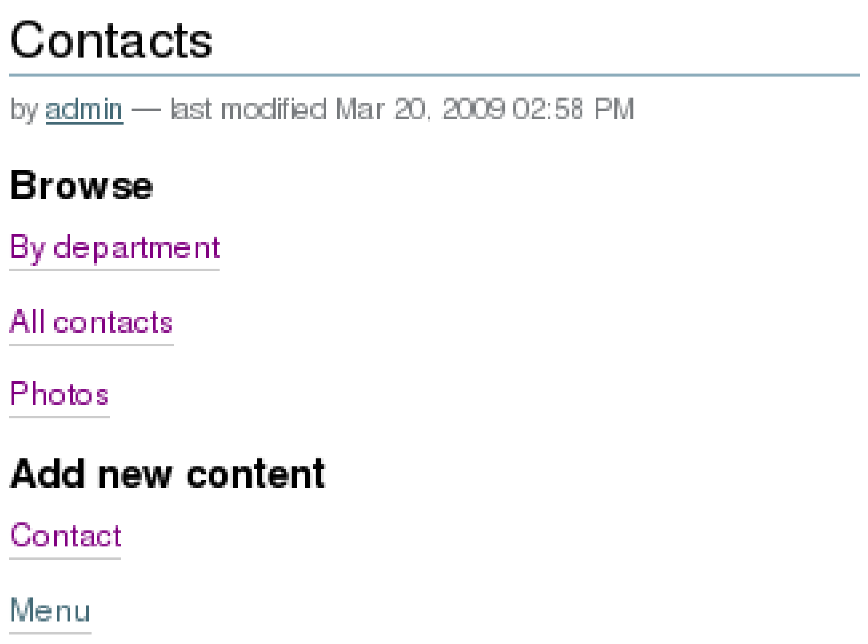
If so, you have to refresh the database. This will re-build the database index entirely, and replace all the previously compiled Plomino formula scripts (the first time a formula is called, it is compiled in a Python Script object in the ZODB).

To do so, go to the database *Design* tab, expand the *Others* section and click on *Database refresh*.

Refresh also migrates your database to your current Plomino version (if Plomino has been upgraded since the database was created).

Start page

By default, the database default screen is the generic database menu:



Contacts

by admin — last modified Mar 20, 2009 02:58 PM

Browse

By department

All contacts

Photos

Add new content

Contact

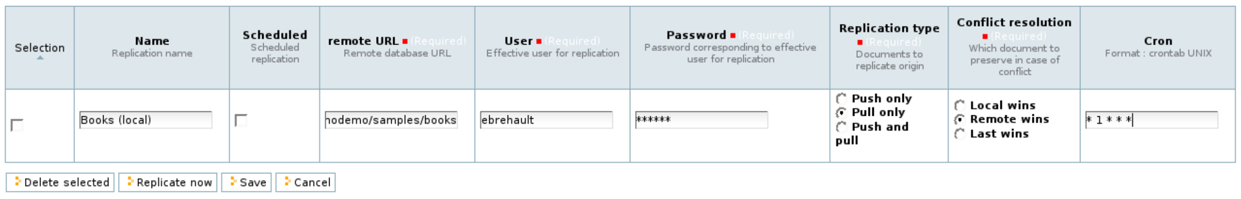
Menu

But you might prefer to display something else instead (for instance a view, a page, a search form, etc.).

In this case, go to your database *Edit* tab, and enter the element id in the *Start page* parameter.

Replication

You can replicate documents between 2 Plomino databases, possibly on 2 different Zope servers.



Selection	Name Replication name	Scheduled Scheduled replication	remote URL ■ (Required) Remote database URL	User ■ (Required) Effective user for replication	Password ■ (Required) Password corresponding to effective user for replication	Replication type ■ (Required) Documents to replicate origin	Conflict resolution ■ (Required) Which document to preserve in case of conflict	Cron Format : crontab UNIX
☐	Books (local)	☐	nodemo/samples/books	ebrehault	******	Push only [✓ Pull only] Push and pull	Local wins [✓ Remote wins] Last wins	* 1 * * *

There are 3 replication modes:

push mode local modifications are replicated on the remote database;

pull mode remote modifications are replicated on the local database;

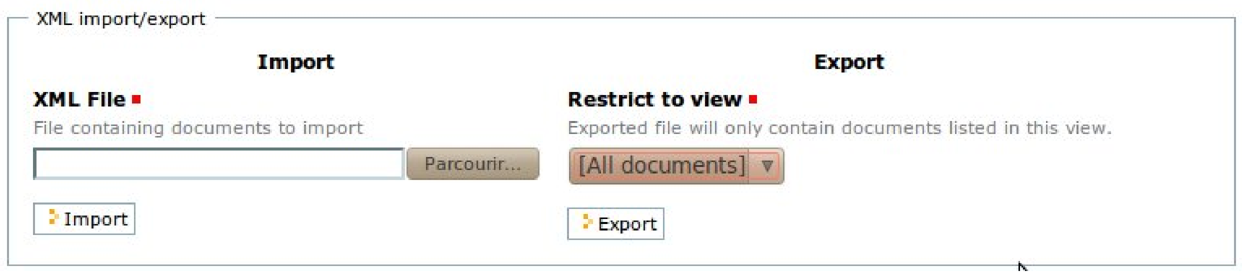
push-pull mode both.

If a document has been modified in both the local and remote databases since the last replication, there are 3 conflict resolution modes: - local wins, - remote wins, - last modified wins.

Replication can be useful to synchronize information between 2 servers, or for mobile workers who want to be able to work on a local replica.

Documents XML import/export

In the *Replication* tab (at the bottom), you can import/export documents from/to an XML file.


XML import/export

Import

XML File
File containing documents to import

Export

Restrict to view
Exported file will only contain documents listed in this view.

[All documents]

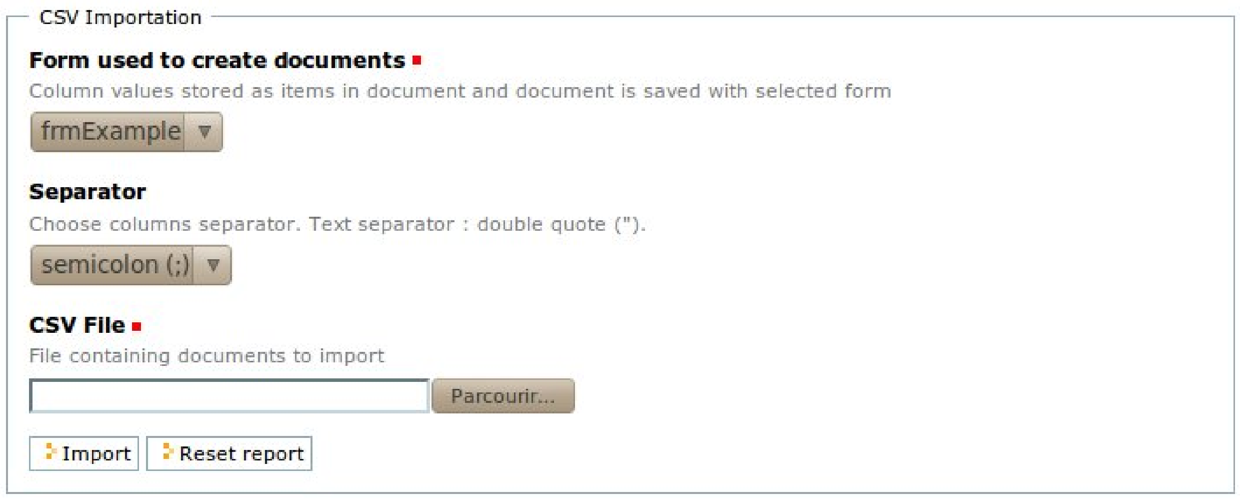
Exported documents can be restricted to a view (meanning that only documents selected in this view will be exported).

Document ids are preserved so if a document already exists in the target database, it is updated and not duplicated.

Note: when importing from XML, the `onSaveDocument` event is not called (as document items are all part of the export).

Documents CSV import

In the *Replication* tab, you can import documents from a CSV file.


CSV Importation

Form used to create documents
Column values stored as items in document and document is saved with selected form

frmExample

Separator
Choose columns separator. Text separator : double quote (").

semicolon (;)

CSV File
File containing documents to import

You need to indicate which form has to be used to create the documents.

The first row in the CSV file must contain the field id for the intended column.

Note: when importing from CSV, the `onSaveDocument` event is called (as some items might needed to be computed) but the index is not refreshed to avoid degrading performance. This means that the index needs to be updated manually, possibly by running an agent that re-saves imported documents on a schedule, or by refreshing the database on a worker ZEO client instance.

Plomino URLs

Database

`OpenDatabase` `http://server/plone/db/OpenDatabase` will open the database home page which either the default home page, either the start page (if defined in the database parameters). Equivalent to:

- `http://server/plone/db`
- `http://server/plone/db/view`

`DatabaseDesign` `http://server/plone/db/DatabaseDesign` will open the database design tab.

`DatabaseACL` `http://server/plone/db/DatabaseACL` will open the database ACL tab.

`DatabaseReplication` `http://server/plone/db/DatabaseReplication` will open the database replication tab.

View

`OpenView` `http://server/plone/db/myview/OpenView` will display the view. Equivalent to:

- `http://server/plone/db/myview`
- `http://server/plone/db/myview/view`

`exportCSV` `http://server/plone/db/myview/exportCSV` will download the view content as a CSV file.

`exportXLS` `http://server/plone/db/myview/exportXLS` will download the view content as an Excel file.

`tojson` `http://server/plone/db/myview/tojson` will return the view content in JSON format.

Form

`OpenForm` `http://server/plone/db/myform/OpenForm` will render the form. Equivalent to:

- `http://server/plone/db/myform`
- `http://server/plone/db/myform/view`

`OpenBareForm` `http://server/plone/db/myform/OpenBareForm` will render the form without the Plone template. It is useful when loading the form through an AJAX call, considering the Plone skin is not needed in that case, and `OpenBareForm` will be more performant.

`searchDocuments` *Only for search forms.* `http://server/plone/db/myform/searchDocuments?field1=value1` will search and display the search results according the parameters.

`tojson` `http://server/plone/db/myform/tojson` will return all the form fields as JSON. `http://server/plone/db/myform/tojson?item=field1` will return the form field `field1` as JSON.

Note: the parameter is named `item` and not `field` in order to expose the same signature as the document `/tojson` URL, so we do not need to test the context in field formulas.

Document

`OpenDocument` `http://server/plone/db/doc1/OpenDocument` will render the document in *read* mode.

Equivalent to:

- `http://server/plone/db/doc1`
- `http://server/plone/db/doc1/view`

`http://server/plone/db/doc1/OpenDocument?openwithform=form1` will render the document in read mode using the specified form.

`EditDocument` `http://server/plone/db/doc1/EditDocument` will render the document in *edit* mode.

Equivalent to `http://server/plone/db/doc1/edit`.

`http://server/plone/db/doc1/EditDocument?openwithform=form1` will render the document in edit mode using the specified form.

`DocumentProperties` `http://server/plone/db/doc1/DocumentProperties` will show all the document information and stored items values.

`AccessControl` `http://server/plone/db/doc1/AccessControl` will show the current access rights and roles in the context of the document.

`delete` `http://server/plone/db/doc1/delete` will delete the document. `http://server/plone/db/doc1/delete?returnurl=an_url` will delete the document and redirect to the specified URL.

`getfile` `http://server/plone/db/doc1/getfile?filename=file1` will download the attached file `file1`.

`deleteAttachment` `http://server/plone/db/doc1/deleteAttachment?fieldname=field1&filename=file1` will delete the attached file `file1` from the field `field1`.

`tojson` `http://server/plone/db/doc1/tojson` will return all the document stored items as JSON.

`http://server/plone/db/doc1/tojson?item=item1` will return the item `item1` as JSON. In the case of a non-stored item (e.g. a *Computed for display* field), its value will be computed using the matching field from the document's form.

`http://server/plone/db/doc1/tojson?item=item1&formid=form1` does the same but the field is explicitly looked up from the `form1` form (which is not necessarily the document's form).

Agent

`runAgent` `http://server/plone/db/agent1/runAgent` will execute the agent.

`runAgent_async` *Requires ``plone.app.async``.* `http://server/plone/db/agent1/runAgent` will execute the agent in asynchronous mode.

Agents

It might be useful to launch the same processing from different places in the application (views action, forms action). To avoid duplicating the code, you can implement the code in an *agent*.

Select *Plomino: agent* in the *Add item* Plone menu, and enter an identifier, a title and the code.

This might be useful to run archiving, cleaning, etc. without giving manager rights to regular users.

By default, an agent runs using the current user access right, but it can also run using the designer (the owner) access right. That way, a regular user might launch an action that would normally require higher privileges if he was doing it manually.

For instance, if an agent is in charge of archiving documents by moving them from the current database to another one, if regular users do not have access to the archive db, they would not be able to put some documents in that db. If the agent is executed as owner, it will not fail.

The agent can be executed (from an action) using the `runAgent()` method:

```
db = plominoDocument.getParentDatabase()
db.MyAgent.runAgent()
```

Note: this method can take `REQUEST` as parameter (this has to be the REQUEST object), which allows variables in the query string to be read and redirection to be controlled (using a `REDIRECT` key on the request).

The agent can also be executed from Python formulas by calling it directly:

```
db = plominoDocument.getParentDatabase()
db.MyAgent('one', 'two', 'three')
```

Note: this method can take optional positional arguments. It does not redirect.

If you install `plone.app.async` on your Zope instance, an agent can also be executed in asynchronous mode.

Resources

A Plomino database contains a `resources` folder in the ZODB which can contain useful extra assets:

- images or icons you may need to insert in your forms;
- CSS or javascript files;
- ZPT templates (see view template below);
- Python scripts, to provide a code library usable from the different formulas (using the `callScriptMethod` method);
- CSV (or other) files containing useful data;
- etc.

To access this folder, go to the *Design* tab, expand the *Others* section and click on *Resources Folder*. It opens the standard *ZMI* screen, which allows new elements to be added.

i18n support

By declaring an i18n domain in the database parameters, Plomino translation will be enabled.

When enabled, any text enclosed by __ will be translated according the defined i18n domain.

It will apply to form layout static content:

```
__What time is it?__
```

would be rendered as:

```
What time is it?
¿qué hora es?
Quelle heure est-il ?
```

(assuming you have an i18n domain containing the msgid "What time is it?" and providing the desired languages)

But it will also apply to any computed field output as well:

```
return context.getItem('the_hour')+" __hours__"
```

would be rendered as:

```
6 hours
6 horas
6 heures
```

If the text does not match any msgid from the i18n domain, it remains unchanged (but without the enclosing __).

The translation mechanism can be called from a formula using the `translate` function provided by PlominoUtils, which can be handy in agents or view columns.

Caching

RAM cache

If your Plomino application contains some time consuming formulas, you can speed up the page display by keeping the result in RAM cache using `getCache` and `setCache`.

Here is an example:

```
result = db.getCache('my_cache_key')
if not result:
    result = make_something_which_cost_CPU(stuff)
    db.setCache('my_cache_key', result)
return result
```

The first time the formula will be called, the `make_something_which_cost_CPU` will be executed, and the result will be put into the cache.

Next time the formula is called, the result is directly read from the cache.

As the cache key is a constant (`my_cache_key`), it will be the same in all the cases (for all the users, in all the pages, etc.).

But of course, the `make_something_which_cost_CPU` function might return a different value depending on the context. If so, you need to produce a cache key that will reflect this context accurately.

For instance, if the result is different according the user, an accurate cache key could be:

```
cache_key = "result_for_"+context.getCurrentUserId()
```

or depending on the document:

```
cache_key = "result_for_"+context.id
```

or anything you might need.

Request cache

Another use case is the repeated usage of a same formula in the same page: sometimes, when rendering a document using a form, several computed fields make the same computation (typical example: you display a table of values, and also a bar chart based on those values).

The code itself can be factorized using a script library in the `/resources` folder, but it will be run twice anyway when rendering the page, and this might impact performance.

Unfortunately, `setCache` and `getCache` might not be relevant because you want the formula to be re-evaluated every time a user displays the page. In that case, you can use `setRequestCache` and `getRequestCache`, so the cache will be associated with the current request, and will only last as long as the request:

```
result = db.getRequestCache('my_cache_key')
if not result:
    result = make_something_which_cost_CPU(stuff)
    db.setRequestCache('my_cache_key', result)
return result
```

Plomino Element Portlet

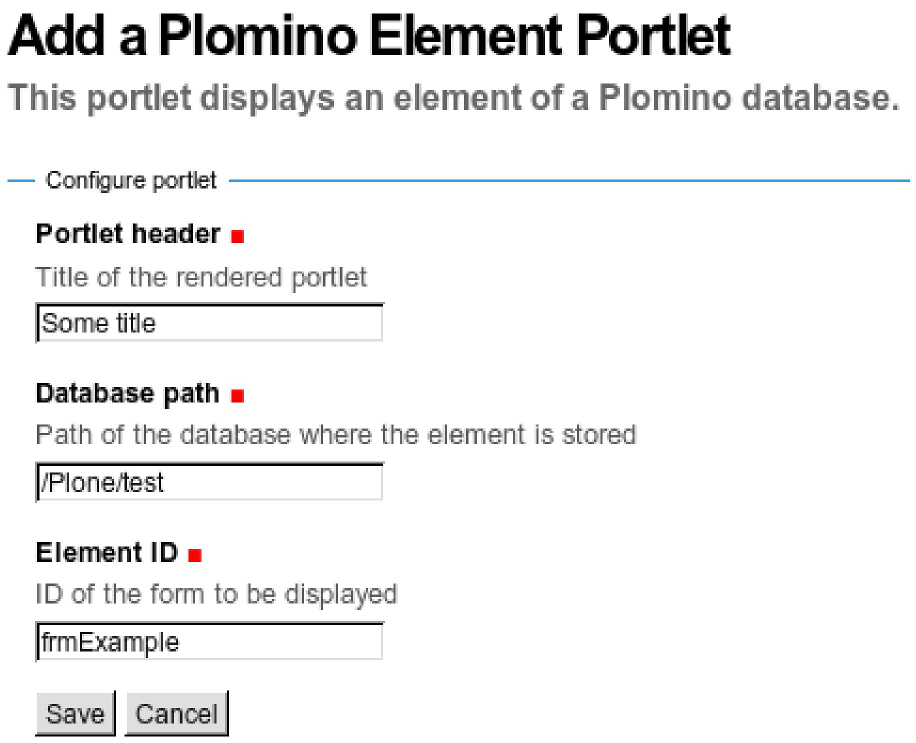
A portlet displaying a Plomino form can be added anywhere in a Plone site. It can be useful to show information, like statistics or charts (thanks to Google Visualization, for example), computed when the page is displayed.

Note: In Plone, when you add a portlet to a page, all of its children pages will contain it too. For example, if you add a portlet to the main page of the site, it will be displayed in every page of the site. You can prevent this mechanism in a child page: click on *Manage Portlets* in this page, find the selector next to the name of the portlet (e.g. *Plomino element portlet*), and select guilabel:*Block*.

You can add a portlet on a page with few steps:

- Click on the link *Manage portlets*
- In the *Add portlet...* selector, choose the *Plomino element portlet* option.

A new page appears, with some fields:



Add a Plomino Element Portlet

This portlet displays an element of a Plomino database.

Configure portlet

Portlet header ■

Title of the rendered portlet

Some title

Database path ■

Path of the database where the element is stored

/Plone/test

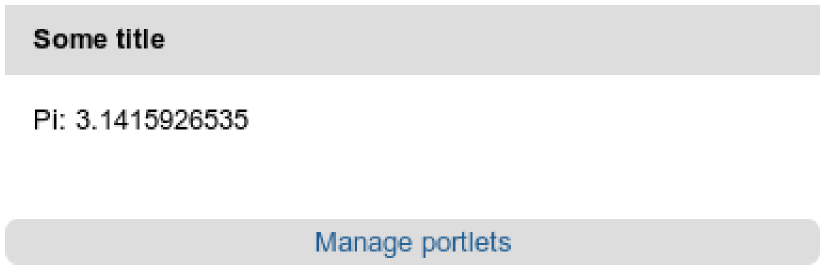
Element ID ■

ID of the form to be displayed

frmExample

- The header field sets the title of the portlet.
- The database path is the path of a Plomino database containing the form to be displayed. If the base is accessible at the URL `http://example.org/Plone/database`, the path is `/Plone/database`. Since there is always an exception to a rule, you have to be careful when the site URLs are re-written (e.g. if the Plone site is behind an Apache server). The path must be the *Plone site* path, not the public URL.
- Element ID is the form identifier (set at its creation) in the database specified previously.

The new portlet is now displayed alongside the page.



Some title

Pi: 3.1415926535

You can control whether the portlet must be displayed or not by adding a field named *Plomino_Portlet_Availability* which formula must return True or False.

Extending Plomino with plugins

Plomino provides a set of utility functions in `PlominoUtils` (`DateToString`, `asUnicode`, etc.).

In addition, custom Plomino utilities can be declared in a custom package, and they will be available from any Plomino formula.

Example:

Create the utility methods in your extension module (e.g. `mypackage.mymodule`):

```
import simplejson as json

def jsonify(obj):
    return json.dumps(obj)

def dejsonify(s):
    return json.loads(s)
```

Create a class to declare them:

```
class MyUtils:
    module = "mypackage.mymodule"
    methods = ['jsonify', 'dejsonify']
```

Declare the module as safe so it can be called from Python Scripts (all Plomino *formula* are Python Scripts):

```
from Products.PythonScripts.Utility import allow_module

allow_module("mypackage.mymodule")
```

And register it with Plomino in a `configure.zcml` file:

```
<utility
      name="MyUtils"
      provides="Products.CMFPlomino.interfaces.IPlominoUtils"
      component="mypackage.mymodule"
      />
```

Now, `jsonify` and `dejsonify` can be used in any Plomino formula.

CHAPTER 10

Plomino class reference

Non-exhaustive list of the classes' methods.

PlominoDatabase

Note: The includes methods inherited from base classes.

`callScriptMethod(self, scriptname, methodname, *args)` Calls a method named `methodname` in a file named `scriptname`, stored in the `resources` folder. If the called method allows it, you may pass some arguments.

`createDocument(self, docid=None)` Returns a new empty document.

`deleteDocument(self, doc)` Delete the document from database.

`deleteDocuments(self, ids=None, massive=True)` Batch delete documents from database. If `massive` is `True`, the `onDelete` formula and index updating are not performed (use `refreshDB` to update).

`getAgent(self, agentid)` Return a PlominoAgent, or None.

`getAgents(self)` Returns all the PlominoAgent objects stored in the database.

`getAllDocuments(self)` Returns catalog brains for all the PlominoDocument objects stored in the database.

`getCurrentMember(self)` Returns the current Plone member.

`getCurrentUserRights(self)` Returns the current user's access rights.

`getCurrentUserRoles(self)` returns the current user's roles.

`getDocument(self, docid)` Returns the PlominoDocument object corresponding to the identifier, or `None`.

`getForm(self, formname)` returns the PlominoForm object corresponding to the identifier.

`getForms(self)` returns all the PlominoForm objects stored in the database.

getIndex(self) returns the PlominoIndex object.

getPortalGroups(self) returns the Plone site groups.

getPortalMembers(self) returns the Plone site members.

getPortalMembersIds(self) returns the Plone site member ids.

getPortalMembersGroupsIds(self) returns the Plone site groups ids and all the Plone site members ids.

getUserRoles(self) returns all the roles declared in the database.

getUsersForRight(self, right) returns the users declared in the ACL and having the given right.

getUsersForRoles(self,role) returns the users declared in the ACL and having the given role.

getView(self, viewname) return the PlominoView object corresponding to the identifier.

getViews(self) returns all the PlominoView objects stored in the database.

hasUserRole(self, userid, role) Returns `True` if the specified user id has the given role.

processImportAPI(self, formName, separator, fileToImport, file_encoding='utf-8')
Import a CSV file to create document using the specified form.

isCurrentUserAuthor(self, doc) returns `True` if the current user is author of the given document or has the PlominoAuthor right.

refreshDB(self) refresh the database index and the formulas.

writeMessageOnPage(self, infoMsg, REQUEST, ifMsgEmpty = '', error = False)
displays a standard Plone status message. The REQUEST parameter is mandatory. Most of the time, `plominoDocument.REQUEST` will be the correct value. `ifMsgEmpty` is the default message to display if `infoMsg` is empty. If `error` is `False`, the message displays as an *informational* message; if `True`, it displays as an *error* message.

PlominoDocument

delete(self, REQUEST=None) deletes the document, and if `REQUEST` contains a key named `returnurl`, uses its value to redirect the client.

deleteAttachment(self, REQUEST) remove file object and update corresponding item value.

getfile(self, filename=None, REQUEST=None) return the file corresponding to the given filename.

getFilenames(self) return the filenames of all the files stored with the document.

getForm(self) returns the form given by the *view form* formula (if the document is opened from a view and if the view has a form formula), else returns the form given by the document's Form item.

getItem(self, name, default='') returns the item value if it exists, else returns the default value (an empty string if not provided).

getItemClassname(self, name) returns the class name of the item .

getItems(self) returns the names of all the items existing in the document.

getParentDatabase(self) Normally used via acquisition by Plomino formulas operating on documents, forms, etc.

getRenderedItem(self, itemname, form=None, convertattachments=False) returns the item value using the rendering corresponding to the field type defined in the form (if form is `None`, it uses the form returned by `getForm()`). If `convertattachments` is `True`, FileAttachments items are converted to text (if possible).

hasItem(self, name) returns `True` if the item exists in the document.

isAuthor(self) returns `True` if the current user is author of the document or has the PlominoAuthor right.

isEditMode(self) returns `True` is the document is being edited, `False` if it is being read. Note the same method is available in PlominoForm, so it can be used transparently in any formula to know if the document is being edited or not.

isNewDocument(self) returns `False` (because an existing document is necessarily not new). Note the same method is available in PlominoForm (and returns `True`), so it can be used transparently in any formula to know if the document is being created or not.

openWithForm(self, form, editmode=False) display the document using the given form's layout (but first, check if the user has proper access rights).

removeItem(self, name) remove the item.

save(self, form=None, creation=False, refresh_index=True) refresh the computed fields and re-index the document in the Plomino index and in the Plone `portal_catalog` (only if `refresh_index` is `True`; `False` might be useful to improve the performance, but a `refreshDatabase` will be needed). It uses the field's formulas defined in the provided form (by default, it uses the form returned by `getForm()`).

send(self, recipients, title, form=None) send the document by mail to the recipients. The document is rendered in HTML using the provided form (by default it uses the form returned by `getForm()`).

setItem(self,name,value) set the value (if the item does not exist, it is created).

PlominoForm

getFormName(self) returns the form id.

getParentDatabase(self) returns the PlominoDatabase object which contains the form.

isEditMode(self) returns `True`.

Note: the same method is available in PlominoDocument, so it can be used transparently in any formula to know if the document is being edit or not.

isNewDocument(self) returns `True` (when the context is a form, it is necessarily a new doc).

Note: the same method is available in PlominoDocument (and returns *False*), so it can be used transparently in any formula to know if the document is being created or not.

PlominoView

exportCSV(self, REQUEST=None) returns the columns values in CSV format. If REQUEST is not `None`, download is proposed to the user.

getAllDocuments(self) returns all the documents which match the Selection Formula. Documents are sorted according the sort column (if defined).

getDocumentsByKey(self, key) returns all documents for which the value of the column used as sort key matches the given key.

getParentDatabase(self) returns the PlominoDatabase object which contains the view.

getViewName(self) returns the view id.

PlominoIndex

dbsearch(self, request, sortindex, reverse=0) searches the documents corresponding to the request (see ZCatalog reference). The returned objects are ZCatalog brains pointing to the documents (see ZCatalog reference).

getKeyUniqueValues(self, key) returns the list of distinct values for an indexed field.

getParentDatabase(self) returns the PlominoDatabase object which contains the index.

refresh(self) refresh the index.

PlominoUtils

Note: PlominoUtils is imported for any formula execution, its methods are always available (importing the module is not needed).

Another module with some useful methods is `Products.PythonScripts.standard`, which can be imported if needed.

actual_context(context, search="PlominoDocument") return the actual context from the request, it will drill into the path until it find a context matching the searched class. Useful in portlet context

actual_path(context) return the actual path from the request. Useful in portlet context

array_to_csv(array, delimiter='\t', quotechar='"') Convert `array` (a list of lists) to a CSV string.

asList(x) If not list, return x in a single-element list. .. note:: If `x` is `None`, this will return `[None]`.

asUnicode(s) Make sure `s` is unicode, decode according to site encoding if needed.

csv_to_array(csvcontent, delimiter='\t', quotechar='"') Convert CSV to array. `csvcontent` may be a string or a file.

DateRange(d1, d2) returns the dates of all the days between the 2 dates.

DateToString(d, format=None, db=None) Converts a date to a string. If `db` is passed, use the database date format.

htmlencode(s) Replaces unicode characters with their corresponding HTML entities.

isDocument(object) Test if the object is a `PlominoDocument`. Useful to distinguish a document context from a form context.

json_dumps(obj) Return the object as a string using the JSON format. Example:

```
>>> json_dumps({"a": [1, 2, "This is a 'quote'"], "b": 0.098098})
'{"a": [1, 2, "This is a \'quote\'"], "b": 0.098098}'
```

json_loads(s) Build an object from a JSON string. Example:

```
>>> json_loads('{"a": [1, 2, "This is a \'quote\'"], "b": 0.098098}')
{u'a': [1, 2, u"This is a 'quote'"], u'b': 0.098098}
```

Log(message, summary='', severity='info', exc_info=False) Write a message to the server event log.

Now() returns current date and time as a DateTime object.

open_url(url, asFile=False, data=None) Load the corresponding url, and retrurn the resulting string (or a stream if asFile is True). If data is not None, it will produce a POST request (and data will be url encoded if it is not a string). IMPORTANT: By default, open_url raises an Unauthorized exception. If the requested domain (note: it might also be a local path) has been declared safe by an local module, it retrieves the content from `url`. To declare a domain as safe:

```
from zope.interface import implements
from zope.component import provideUtility
from Products.CMFPlomino.interfaces import IPlominoSafeDomains

class MySafeDomains:
    implements(IPlominoSafeDomains)

    domains = [
      "http://api.geonames.org",
      "/var/public"
    ]
provideUtility(MySafeDomains, IPlominoSafeDomains)
```

PlominoTranslate(message, context, domain='CMFPlomino') translate the given message using the Plone i18n engine (using the given domain).

sendMail(db, recipients, title, message_in, sender=None, cc=None, bcc=None, immediate=False
Send a mail to the recipients. If sender is None, it will use the current user mail address. By default it accepts a html message, you can submit text instead and supply msg_format='text'.

StringToDate(str_d, format='%Y-%m-%d', db=None) Converts a string to a date. If `db` is passed, use the database date format. If `format=None`, guess.

PlominoTranslate(msgid, context, domain='CMFPlomino') Look up the translation for `msgid` in the current language.

urlencode(h) Convert a dictionary into a URL querystring (a `key=value&` string). Example:

```
>>> urlencode({"option": 5, "article": "9879879"})
'article=9879879&option=5'
```

urlquote(string) Replace special characters in a string using the `%xx` escape. Example:

```
>>> urlquote('runAgent?REDIRECT=True&action=accept')
'runAgent%3FREDIRECT%3DTrue%26action%3Daccept'
```

userFullname(db, userid) returns the user full name.

userInfo(db, userid) returns the Member object corresponding to the user id (it may be used to get the user email address for instance).

PlominoAgent

`getParentDatabase(self)` returns the PlominoDatabase object which contains the agent.

`runAgent(self, REQUEST=None)` runs the agent. If REQUEST is provided, there is a redirection to the database home page, unless the REQUEST contains a REDIRECT key If so, the formula returned value is used as the redirection URL.

`__call__(*args)` if agents are called from Python code, they can take positional arguments.

CHAPTER 11

Indices and tables

Glossary

This is a glossary for Plomino-specific terms. It is still heavily under construction.

Action An Action in Plomino renders a UI element that allows the user to trigger some processing.

Agent An Agent object executes some code as a configured user.

Column A Column defines a value that should be computed for each *Document* that matches a *View*.

Content Rules A Plone feature allowing code or events to be configures upon events such as object creation or modification.

Document A Document in Plomino contains named *Item* objects, that are rendered by *View* and *Form* objects.

Field A Field object, which lives in a Form, renders a Widget, sometimes accepts input, and triggers events.

Form A Form object, containing fields, edits *Document* objects according to the fields and events configured on the form.

Formula In the context of Plomino, a *formula* is a snippet of restricted Python code executed in the context of a field or event.

Hide-when A *formula* that determines whether an action or section of a form is rendered.

Item An Item stored a value on a *Document*.

Sub-form A Form object rendered as part of another form.

View A View defines a group of Documents related by a *selection formula*.

ZMI The Zope Management Interface.

ZODB The Zope Object Database: transparent persistence for Python objects, since 2002. (Before 2002, it was part of Zope.)

Automatically generated TODO list

The following list is automatically generated from `.. TODO::` directives in the text.

See http://sphinx.pocoo.org/ext/todo.html for details.

TODO items

- genindex
- modindex
- search

Index