django-generic-bookmarks Documentation

Release 0.2

Francesco Banconi

Oct 05, 2017

Contents

This application provides bookmarks management functionality to a Django project.

For instance, using bookmarks, users can store their favourite contents, or items they follow, or topics they like or dislike.

A bookmark connects users to Django contents in a generic way, without modifying existing models tables, and *django-generic-bookmarks* exposes a simple API to handle them, yet allowing the management of bookmarks in complex scenarios too.

The bookmarks can be stored using different backends. The default one uses Django models to store user's preferences in the database, but it is possible to write customized backends, and the application, out of the box, includes also a MongoDB backend.

The source code for this app is hosted on https://bitbucket.org/frankban/django-generic-bookmarks

CHAPTER 1

Contents

Getting started

Requirements

Python	>= 2.5
Django	>= 1.3

jQuery >= 1.4 is required if you want to take advantage of *AJAX* features described above and in *Templatetags reference*.

`pip install mongoengine` is needed if you want to use the MongoDB backend.

Installation

The Mercurial repository of the application can be cloned with this command:

```
hg clone https://bitbucket.org/frankban/django-generic-bookmarks
```

The `bookmarks` package, included in the distribution, should be placed on the `PYTHONPATH`.

Otherwise you can just `pip install django-generic-bookmarks`.

Configuration

Add the request context processor in your *settings.py*, e.g.:

```
from django.conf.global_settings import TEMPLATE_CONTEXT_PROCESSORS
TEMPLATE_CONTEXT_PROCESSORS += (
    'django.core.context_processors.request',
)
```

Add `'bookmarks'` to the `INSTALLED_APPS` in your *settings.py*.

The application, by default, uses Django models to save bookmarks in the database. If you want to use MongoDB instead, just add in your *settings.py*:

```
GENERIC_BOOKMARKS_BACKEND = 'bookmarks.backends.MongoBackend'
GENERIC_BOOKMARKS_MONGODB = {"NAME": "bookmarks"}
```

See *Customization* section in this documentation for other settings options and *Backends reference* for a detailed description of provided backends.

Add the bookmarks urls to your *urls.py*, e.g.:

```
(r'^bookmarks/', include('bookmarks.urls')),
```

Time to create the needed database tables using *syncdb* management command:

```
./manage.py syncdb
```

Quickstart

To allow a user to bookmark a Django model instance, the model must be registered as *bookmarkable*, i.e. the system must know that instances of that model can be bookmarked by users.

For example, if you have an *Article* model and you want users to add articles to their favourites, you must register the model as bookmarkable, e.g.:

```
from bookmarks.handlers import library
library.register(Article)
```

You can register models anywhere you like. However, you'll need to make sure that the module it's in gets imported early on so that the model gets registered before any bookmark is saved by the user. This makes your app's *models.py* a good place to put the above code.

Under the hood you have registered the *Article* model with a default bookmark handler. Handlers are Python classes encapsulating bookmarking options for a given model, while *library* is a singleton registry that stores handlers. For a detailed explanation see *Using handlers*.

Now it's time to let your users add an article to his favourites, and this is possible using one of the provided template-tags. In the code below we assume that *article* is the *Article* model instance.

```
{% load bookmarks_tags %}

{% bookmark_form for article %}
```

This code snippet just displays a form to add or remove the article from user's favourites.

AJAX is also supported using jQuery, e.g.:

```
{% load bookmarks_tags %}

<script src="path/to/jquery.js" type="text/javascript"></script>
<script src="{{ STATIC_URL }}bookmarks/bookmarks.js" type="text/javascript"></script>

{% bookmark_form for article %}
```

It is possible to get the form as a template variable in the current context instead of displaying it. This way we can customize the way the form is presented, e.g.:

```
{% bookmark_form for article as form %} {# <-- note the 'as' argument #}

<script src="path/to/jquery.js" type="text/javascript"></script>
<script src="{{ STATIC_URL }}bookmarks/bookmarks.js" type="text/javascript"></script>

{% if form %}
    {% if user.is_authenticated %}
        <form action="{% url bookmarks_bookmark %}" method="post" accept-charset="UTF-
↪8" class="bookmarks_form">
            {% csrf_token %}
            {{ form }}
            {% with form.bookmark_exists as exists %}
                {# another hidden input is created to handle javascript submit event
↪#}
                <input class="bookmarks_toggle" type="submit" value="add"{% if exists
↪%} style="display: none;"{% endif %}/>
                <input class="bookmarks_toggle" type="submit" value="remove"{% if not␣
↪exists %} style="display: none;"{% endif %}/>
            {% endwith %}
            <span class="error" style="display: none;">Error during process</span>
        </form>
    {% else %}
        Handle anonymous users.
    {% endif %}
{% endif %}
```

This application provides other templatetags (e.g.: for bookmarks retreival) and the `bookmark_form` has other useful options, explained in detail in *Templatetags reference*.

Note that the form template variable will be *None* if:

- the user is not authenticated
- the instance is not bookmarkable
- the key is not allowed

What is a key? It is a way to define different kind of bookmarks. For example, a user can add the article to his liked or to his disliked, and so we need a key to tell the system what he is doing. But this is an argument for the next section: *Using handlers*.

Using handlers

As seen in *Getting started*, a model instance can be bookmarked by users only if its model class is handled. Being handled, for a model, means it is registered with a bookmarks handler.

We have seen how to do that:

```
from bookmarks.handlers import library
library.register(MyModel)
```

The handler class is an optional argument of the `library.register` method, and, if not provided, the default `bookmarks.handlers.Handler` is used.

The previous code can be written:

```
from bookmarks.handlers import library, Handler
ratings.register(MyModel, Handler)
```

For convenience, `library.register` can also accept a list of model classes in place of a single model; this allows easier registration of multiple models with the same handler class, e.g.:

```
from bookmarks.handlers import library, Handler
ratings.register([Article, BlogEntry], Handler)
```

You can register models anywhere you like. However, you'll need to make sure that the module it's in gets imported early on so that the model gets registered before any bookmark is saved by the user. This makes your app's *models.py* a good place to put the above code.

Handlers are Python classes encapsulating bookmarking options for a given model, and theese options can be overridden while registering a model, e.g.:

```
from bookmarks.handlers import library, Handler
ratings.register(MyModel, Handler,
    allowed_keys=['likes', 'dislikes'], form_class=MyCustomForm)
```

Three things are done in the code snippet above:

- *MyModel* is registered as a bookmarkable model, i.e. users can save instances of that model as bookmarks.
- Two types of bookmarks are allowed: *likes* and *dislikes*. This means that users can like or dislike *MyModel* instances (note that keys are just arbitrary strings)
- *MyCustomForm* will be used to save bookmarks (in place of the form provided by the application)

See *Handlers reference* for a list of all available handlers options.

Later it is possible to retreive the handler instance used to manage bookmarks for a particular model or instance:

```
from bookmarks.handlers import library
# handler instance for article class
handler = library.get_handler(article)
# handler instance for MyModel
handler = library.get_handler(MyModel)
```

Custom Handlers

There are situations where the built-in options are not sufficient.

What if, for instance, you want to use different forms for staff and normal users?

As in Django own `contrib.admin.ModelAdmin`, you can write subclasses of `bookmarks.handlers.Handler` to override the methods which actually perform the bookmark process, and apply any logic you desire.

Here is an example meeting the needs described above:

```
from bookmakrs.handlers import library, Handler

class MyHandler(Handler):

    def get_form_class(self, request):
        """
        Return the form class that will be used to add or remove bookmarks.
        Default is *self.form_class*.
        """
        return StaffForm if request.user.is_staff else self.form_class

library.register(MyModel, MyHandler)
```

Examples of handler customizations can be found in *Usage and examples*.

Handlers API

As seen in *Getting started*, you can let users add or remove bookmarks using a simple templatetag:

```
{% load bookmarks_tags %}

{% bookmark_form for article %}
```

But what happens when the user clicks to add or remove a bookmark?

The handler is used to do the real work.

1. Key management

Initially the handler is responsable of producing a valid bookmark *key*.

The key is an arbitrary string representing the type of bookmark we are saving. For example, users can like an article or hate it, or maybe they want to be notified on comments of that article. Theese are different types of bookmarks and can be expressed using different keys (e.g.: `likes`, `hates`, `comments`).

The two methods called to handle keys are:

get_key(*self*, *request*, *instance*, *key=None*)
Return the bookmark key to be used to save the bookmark of *instance*.

Subclasses can return different keys based on the *request*, on the given target object *instance* or the optional *key* that can be provided for example by the templatetags.

Here is an example of a templatetag providing a key:

```
{% load bookmarks_tags %}
{% bookmark_form for article using 'favourite' %}
```

For example, if you want a different key to be used if the user is staff, you can override this method in this way:

```
def get_key(self, request, instance, key=None):
    return 'staff' if request.user.is_superuser else 'normal'
```

If you do not customize things, this method returns the given *key* (if not *None*) or a default key `main`.

allow_key(*self*, *request*, *instance*, *key*)
This method is called when the user tries to bookmark an object using the given bookmark *key* (e.g. when the bookmark view is called with POST data).

The bookmarking process continues only if this method returns True (i.e. a valid key is passed).

For example, if you want two different bookmarks for each target object, you can use two forms (each providing a different key, say 'main' and 'other') and then allow those keys:

```
def allow_key(self, request, instance, key):
    return key in ('main', 'other')
```

By default this method allows keys listed in *self.allowed_keys*.

See *Usage and examples* for a deeper explanation of how to handle keys.

2. Bookmark saving

Five handlers methods are involved in bookmarks saving:

get_form(*self*, *request*, ***kwargs*)
that returns the form that actually adds or remove a bookmark, and that calls...

get_form_class(*self*, *request*)
to get the form class used (usually is *Handler.form_class* that by default points to *bookmarks.forms.BookmarkForm*).

pre_save(*self*, *request*, *form*)
Called just before the bookmark is added or removed, this method takes the *request* and the *form* instance.

Subclasses can use this method to check if the bookmark can be saved or deleted, and, if necessary, block the bookmarking process returning False.

This method is called by a *signals.bookmark_pre_save* receiver always attached to the handler by the registry.

It's up to the developer if override this method or just connect another listener to the signal: the bookmarking process is killed if just one receiver returns False.

save(*self*, *request*, *form*)
Save the bookmark to the database. Return the saved bookmark.

post_save(*self*, *request*, *bookmark*, *added*)
Called just after a bookmark is added or removed.

The given arguments are the current *request*, the just added or deleted *bookmark* and the boolean *added* (True if the bookmark was added).

This method is called by a *signals.bookmark_post_save* receiver always attached to the handler by the registry.

It's up to the developer if override this method or just connect another listener to the signal.

By default, this method does noting.

3. HTTP Response

Finally, the reponse to the client is managed by

response(*self*, *request*, *bookmark*, *created*)
that, by default, calls...

ajax_response(*self*, *request*, *bookmark*, *created*)
Called if the request is ajax. Return a JSON reponse containing:

```
{
    'key': 'the_bookamrk_key',
    'bookmark_id': bookmark.id,
    'user_id': <the id of the bookmarker>,
    'created': <True if bookmark is created, False otherwise>,
}
```

or

normal_response(*self*, *request*, *bookmark*, *created*)
Called by *self.response* when the request is not ajax. Return a redirect response.

While the complete handlers API is described in *Handlers reference*, maybe now it's time to read *Usage and examples*.

Usage and examples

As seen previously in *Using handlers*, we can customize the bookmark process creating and registering bookmarks handlers.

In this section we will deal with some real-world examples of usage of Django Generic Bookmarks.

Simple bookmarks

As seen in *Getting started*, adding bookmarks functionality to a Django project is straightforward.

It is only needed to register bookmarkable models (*Article* in our example):

```
from bookmarks.handlers import library
library.register(Article)
```

and then to display the form using a templatetag, having *article* as an *Article* model instance:

```
{% load bookmarks_tags %}

{% bookmark_form for article %}
```

Multiple types of bookmarks

Assume we want users to follow site contents and/or to share them. This means that we want two types of bookmark for a single model instance: let' say `followed` and `shared`.

First, we have to register a model informing the system that the two keys above are allowed:

```
from bookmarks.handlers import library, Handler
ratings.register(MyModel, Handler, allowed_keys=['followed', 'shared'])
```

And then, in the template, we get one bookmark form for each key:

```
{# follow/unfollow #}

{% bookmark_form for article using 'followed' as follow_form %} {# <-- note the 'using
↪' argument #}

{% if follow_form %}
    {% if user.is_authenticated %}
        <form action="{% url bookmarks_bookmark %}" method="post" accept-charset="UTF-
↪8" class="bookmarks_form">
            {% csrf_token %}
            {{ follow_form }}
            {% with follow_form.bookmark_exists as exists %}
                {# another hidden input is created to handle javascript submit event
↪#}
                <input class="bookmarks_toggle" type="submit" value="follow"{% if␣
↪exists %} style="display: none;"{% endif %}/>
                <input class="bookmarks_toggle" type="submit" value="stop following"{
↪% if not exists %} style="display: none;"{% endif %}/>
            {% endwith %}
            <span class="error" style="display: none;">Errors during process</span>
        </form>
    {% else %}
```

```
        Handle anonymous users.
    {% endif %}
{% endif %}

{# share/unshare #}

{% bookmark_form for article using 'shared' as share_form %} {# <-- note the 'using'␣
↪argument #}

{% if share_form %}
    {% if user.is_authenticated %}
        <form action="{% url bookmarks_bookmark %}" method="post" accept-charset="UTF-
↪8" class="bookmarks_form">
            {% csrf_token %}
            {{ share_form }}
            {% with share_form.bookmark_exists as exists %}
                {# another hidden input is created to handle javascript submit event
↪#}
                <input class="bookmarks_toggle" type="submit" value="share"{% if␣
↪exists %} style="display: none;"{% endif %}/>
                <input class="bookmarks_toggle" type="submit" value="unshare"{% if␣
↪not exists %} style="display: none;"{% endif %}/>
            {% endwith %}
            <span class="error" style="display: none;">Errors during process</span>
        </form>
    {% else %}
        Handle anonymous users.
    {% endif %}
{% endif %}
```

Note that we are using two submit inputs for each form, and all of them have *bookmarks_toggle* html class: this is not required, but it makes easier for a Javascript to show and hide them based on AJAX request, as described below.

See *Forms reference* to know more about forms, and *Templatetags reference* for further explanation about provided templatetags.

Conditional bookmarks

Assume we want the system to automatically assign a key to bookmarks based on some conditions.

For example, we want users to express an interest for not yet released films, or to like them when they finally are on theaters.

So we need to switch between two keys (let's say `interests` and `likes`) based on release status of the film:

```
import datetime
from bookmarks.handlers import library, Handler

class FilmHandler(Handler):

    allowed_keys = ('interests', 'likes')

    def get_key(self, request, instance, key=None):
        if key is None:
            today = datetime.date.today()
            key = 'interests' if instance.release_date < today else 'likes'
        return key
```

```
library.register(Film, FilmHandler)
```

Nothing remains but to retreive the form in the template without specifying the key to use.

Add/remove bookmarks using links

Sometimes you may want to use links instead of submit inputs to let users add or remove bookmarks.

This is achievable using a little bit of Javascript, and jQuery of course:

```
{% bookmark_form for article as form %} {# <-- note the 'using' argument #}

{% if form %}
    {% if user.is_authenticated %}
        <form action="{% url bookmarks_bookmark %}" method="post" accept-charset="UTF-
↪8" class="bookmarks_form">
            {% csrf_token %}
            {{ form }}
            {% with form.bookmark_exists as exists %}
                <span class="bookmarks_toggle"{% if exists %} style="display:none"{%␣
↪endif %}>
                    <a rel="nofollow" href="javascript:void(0)" onclick="$(this).
↪closest('form').submit();">Follow</a>
                </span>
                <span class="bookmarks_toggle"{% if not exists %} style="display:none"
↪{% endif %}>
                    <a rel="nofollow" href="javascript:void(0)" onclick="$(this).
↪closest('form').submit()">Stop following</a>)
                </span>
            {% endwith %}
            <span class="error" style="display: none;">Errors during process</span>
        </form>
    {% else %}
        Handle anonymous users.
    {% endif %}
{% endif %}
```

This is only an example of how to submit a form using the *onclick* event of a link.

Using AJAX

In all the examples seen above, the form is used with some tricks:

- the form class is *bookmarks_form*
- we use two elements to submit the form, one for adding and one for removing a bookmark, and one of them is deactivated (not displayed)
- theese two elements have *bookmarks_toggle* html class
- there is a hidden element with class *error*

They are really needed only if you want to use AJAX in the bookmark process loading in the template jQuery and the provided *bookmarks.js*, e.g.:

```
{% bookmark_form for article as form %}

<script src="path/to/jquery.js" type="text/javascript"></script>
<script src="{{ STATIC_URL }}bookmarks/bookmarks.js" type="text/javascript"></script>

...
```

The Javascript performs various operations:

- POST data to the server using AJAX
- toggle the elements having *bookmarks_toggle* html class
- if errors occurs during process, show the element having *error* class
- trigger the `bookmarked` event on the form, with data returned by the server, e.g.:

```
{
    'key': 'the_bookamrk_key',
    'bookmark_id': bookmark.id,
    'user_id': <the id of the bookmarker>,
    'created': <True if bookmark is created, False otherwise>,
}
```

Performance and database denormalization

One goal of *Django Generic Bookmarks* is to provide a generic solution to connect model instances to users without the need to edit your (or third party) models.

Sometimes, however, you may want to denormalize data, for example because you need to minimize queries for tables with a lot of records, or for backward compatibility with legacy code.

Assume you want to store the bookmarks count for your model instances. For example, we want to store the number of users who liked an article.

This is easily achievable, again, customizing the handler, e.g.:

```
from bookmarks.handlers import library, Handler

class ArticleHandler(Handler):

    def post_save(self, request, bookmark, added):
        if bookmark.key == 'likes':
            count = self.backend.filter(key=bookmark.key).count()
            instance = bookmark.content_object
            instance.num_likes = count
            instance.save()

library.register(Article, ArticleHandler)
```

Bookmarks and cache

See **ajax_bookmark_form** in *Templatetags reference*.

Retreiving bookmarks

The backend used to store and retreive bookmarks is always accessible from the *library* registry.

While a complete description of backends can be found in *Backends reference*, here is a brief summary of the API:

```
from bookmarks.handlers import library

# get all bookmarks saved by a user
bookmarks = library.backend.filter(user=user)

# get all bookmarks of a specified instance and key
bookmarks = library.backend.filter(instance=article, key='likes')

# get all articles bookmarks
bookmarks = library.backend.filter(model=Article)

# add/remove bookmarks
bookmark = library.backend.add(user, article, 'likes')
bookmark = library.backend.remove(user, article, 'likes')

# get a bookmark
bookmark = library.backend.get(user, article, 'likes')

# check for bookamrk existance
exists = library.backend.exists(user, article, 'likes')
```

Note that backend is also present as an attribute of handlers, e.g.:

```
from bookmarks.handlers import library
handler = library.get_handler(Article)
backend = handler.backend
```

It is easy to retreive bookmarks in templates using the **bookmark** and **bookmarks** templatetags (see *Templatetags reference*).

Annotating user's bookmarks

See **annotate_bookmarks** function in *Models reference*.

Deleting model instances

To preserve database integrity, when you delete a model instance all related bookmarks are contextually deleted too.

Customization

When you register an handler you can customize all the bookmark options, as seen in *Using handlers*.

But it is also possible to register an handler without overriding options or methods, and that handler will work using pre-defined global settings.

This section describes the settings used to globally customize bookmark handlers, together with their default values.

`GENERIC_BOOKMARKS_BACKEND = None`

default bookmark model (if None, *bookmarks.backends.ModelBackend* is used)

to use MongoDB backend you can just write:

```
GENERIC_BOOKMARKS_BACKEND = 'bookmarks.backends.MongoBackend'
```

`GENERIC_BOOKMARKS_DEFAULT_KEY = 'main'`

default key to use for bookmarks when there is only one bookmark-per-content

---

`GENERIC_BOOKMARKS_NEXT_QUERYSTRING_KEY = 'next'`

querystring key that can contain the url of the redirection performed after adding or removing bookmarks

---

`GENERIC_BOOKMARKS_CAN_REMOVE_BOOKMARKS = True`

set to False if you want to globally disable bookmarks deletion

---

`GENERIC_BOOKMARKS_MONGODB = {'NAME': '', 'USERNAME': '', 'PASSWORD': '', 'PARAMETERS': {}}`

mongodb backend connection parameters

if the instance of MongoDB is executed in localhost without authentication you can just write:

```
GENERIC_BOOKMARKS_MONGODB = {"NAME": "bookmarks"}
```

Templatetags reference

In order to use the following templatetags you must `{% load bookmarks_tags %}` in your template.

bookmark_form

bookmarks.templatetags.bookmarks_tags.**bookmark_form**(*parser*, *token*)

Return, as html or as a template variable, a Django form to add or remove a bookmark for the given instance and key, and for current user.

Usage:

```
{% bookmark_form for *instance* [using *key*] [as *varname*] %}
```

The key can be given hardcoded (surrounded by quotes) or as a template variable. Note that if the key is not given, it will be generated using the handler's *get_key* method, that, if not overridden, returns the default key.

If the *varname* is used then it will be a context variable containing the form. Otherwise the form is rendered using the first template found in the order that follows:

```
bookmarks/[app_name]/[model_name]/[key]/form.html
bookmarks/[app_name]/[model_name]/form.html
bookmarks/[app_name]/[key]/form.html
bookmarks/[app_name]/form.html
bookmarks/[key]/form.html
bookmarks/form.html
```

The *app_name* and *model_name* refer to the instance given as argument to this templatetag.

Example:

```
{% bookmark_form for myinstance using 'mykey' as form %}

{% if form %}
    {% if user.is_authenticated %}
        <form action="{% url bookmarks_bookmark %}" method="post" accept-charset=
↪"UTF-8" class="bookmarks_form">
            {% csrf_token %}
            {{ form }}
            {% with form.bookmark_exists as exists %}
                {# another hidden input is created to handle javascript submit␣
↪event #}
                <input class="bookmarks_toggle" type="submit" value="add"{% if␣
↪exists %} style="display: none;"{% endif %}/>
                <input class="bookmarks_toggle" type="submit" value="remove"{% if␣
↪not exists %} style="display: none;"{% endif %}/>
            {% endwith %}
            <span class="error" style="display: none;">Error during process</span>
        </form>
    {% else %}
        Handle anonymous users.
    {% endif %}
{% endif %}
```

The template variable (or the html) will be None if:

- the user is not authenticated
- the instance is not bookmarkable
- the key is not allowed

AJAX is also supported using jQuery, e.g.:

```
{% load bookmarks_tags %}

<script src="path/to/jquery.js" type="text/javascript"></script>
<script src="{{ STATIC_URL }}bookmarks/bookmarks.js" type="text/javascript"></
↪script>

{% bookmark_form for article %}
```

ajax_bookmark_form

bookmarks.templatetags.bookmarks_tags.**ajax_bookmark_form**(*parser*, *token*)

Use this just like the *bookmark_form* templatetag. The only difference here is that it always render a form template (so you can't use the *as varname* part), and the form template is rendered using an AJAX request.

This is useful for example when you want to show add/remove bookamrk interaction for authenticated users even in a cached template.

You need to load jQuery before using this templatetag.

bookmark

bookmarks.templatetags.bookmarks_tags.**bookmark**(*parser*, *token*)

Return as a template variable a bookmark object for the given instance and key, and for current user.

Usage:

```
{% bookmark for *instance* [using *key*] as *varname* %}
```

The key can be given hardcoded (surrounded by quotes) or as a template variable. Note that if the key is not given, it will be generated using the handler's *get_key* method, that, if not overridden, returns the default key.

The template variable will be None if:

- the user is not authenticated
- the instance is not bookmarkable
- the bookmark does not exist

bookmarks

bookmarks.templatetags.bookmarks_tags.**bookmarks**(*parser*, *token*)

Return as a template variable all bookmarks, with possibility to filter them by user, or to take only bookmarks of a particular model and using a specified key. It is also possible to reverse the order of bookmarks (by default they are ordered by date).

Usage:

```
{% bookmarks [of *model*] [by *user*] [using *key*] [reversed] as *varname* %}
```

Examples:

```
{# get all bookmarks saved by myuser #}
{% bookmarks by myuser as myuser_bookmarks %}

{# get all bookmarks for myinstance using mykey #}
{% bookmarks of myinstance using *mykey* as bookmarks %}

{# getting all bookmarks for model 'myapp.mymodel' in reverse order #}
{% bookmarks of 'myapp.mymodel' reversed as varname %}
```

Note that the args *model* can be:

•a model name as string (e.g.: 'myapp.mymodel')

•a model instance

The key can be given hardcoded (surrounded by quotes) or as a template variable.

Handlers reference

Default handler

class bookmarks.handlers.**Handler**

Encapsulates content bookmarking options for a given model.

This class can be subclassed to specify different behaviour and options for bookmarks of a given model, but can also be used directly, just to handle default any model using default options.

The default handler uses the project's settings as options: this way you can register not customized handlers and then modify their options just editing the settings file.

Most common bookmarking needs can be handled by subclassing *Handler* and changing the values of predefined attributes. The full range of built-in options is as follows.

default_key
default key to use for bookmarks when there is only one bookmark-per-content (default: *'main'*)

allowed_keys
the bookmark allowed keys (default: *['main']*)

next_querystring_key
querystring key that can contain the url of the redirection performed after bookmarking (default: *'next'*)

can_remove_bookmarks
set to False if you want to globally disable bookmarks deletion (default: *True*)

form_class
form class that will be used to handle bookmark's adding and removing (default: *bookmarks.forms.BookmarkForm*)

For situations where the built-in options listed above are not sufficient, subclasses of *Handler* can also override the methods which actually perform the bookmarking process, and apply any logic they desire.

See the method's docstrings for a description of how each method is used during the bookmarking process.

get_key(*self*, *request*, *instance*, *key=None*)
Return the bookmark key to be used to save the bookmark.

Subclasses can return different keys based on the *request*, on the given target object *instance* or the optional *key* (that can be provided for example by the templatetags).

For example, if you want a different key to be used if the user is staff, you can override this method in this way:

```
def get_key(self, request, instance, key=None):
    return 'staff' if request.user.is_superuser else 'normal'
```

allow_key(*self*, *request*, *instance*, *key*)
This method is called when the user tries to bookmark an object using the given bookmark *key* (e.g. when the bookmark view is called with POST data).

The bookmarking process continues only if this method returns True (i.e. a valid key is passed).

For example, if you want two different bookmarks for each target object, you can use two forms (each providing a different key, say 'main' and 'other') and then allow those keys:

```
def allow_key(self, request, instance, key):
    return key in ('main', 'other')
```

By default this method allows keys listed in *self.allowed_keys*.

get_form_class (*self*, *request*)
Return the form class that will be used to add or remove bookmarks. Default is *self.form_class*.

get_form (*self*, *request*, ***kwargs*)
Return an instance of the form, using given *request*, the backend currently used by the handler and all given *kwargs*.

pre_save (*self*, *request*, *form*)
Called just before the bookmark is added or removed, this method takes the *request* and the *form* instance.

Subclasses can use this method to check if the bookmark can be saved or deleted, and, if necessary, block the bookmarking process returning False.

This method is called by a *signals.bookmark_pre_save* receiver always attached to the handler by the registry.

It's up to the developer if override this method or just connect another listener to the signal: the bookmarking process is killed if just one receiver returns False.

save (*self*, *request*, *form*)
Save the bookmark to the database. Must return the saved bookmark.

post_save (*self*, *request*, *bookmark*, *added*)
Called just after a bookmark is added or removed.

The given arguments are the current *request*, the just added or deleted *bookmark* and the boolean *added* (True if the bookmark was added).

This method is called by a *signals.bookmark_post_save* receiver always attached to the handler by the registry.

It's up to the developer if override this method or just connect another listener to the signal.

By default, this method does noting.

ajax_response (*self*, *request*, *bookmark*, *created*)
Called by *self.response* when the request is ajax. Return a JSON reponse containing:

```
{
    'key': 'the_bookamrk_key',
    'bookmark_id': bookmark.id,
    'user_id': <the id of the bookmarker>,
    'created': <True if bookmark is created, False otherwise>,
}
```

normal_response (*self*, *request*, *bookmark*, *created*)
Called by *self.response* when the request is not ajax. Return a redirect response.

response (*self*, *request*, *bookmark*, *created*)
Callback used by the bookmarking views, called when the user successfully added or removed a bookmark.

Must return a Django http response (usually a redirect, or some json if the request is ajax).

The real job is done in the *ajax_response* and *normal_response* methods above.

fail (*self*, *request*, *errors*)
Callback used by the bookmarking views, called when bookmark form did not validate. Must return a Django http response.

remove_all_for (*self*, *sender*, *instance*, ***kwargs*)
The target object *instance* of the model *sender*, is being deleted, so we must delete all the bookmarks related to that instance.

This receiver is usually connected by the bookmark registry, when a handler is registered.

Library

class bookmarks.handlers.**Registry**

Registry that stores the handlers for each content type bookmarks system.

An instance of this class will maintain a list of one or more models registered for being bookmarked, and their associated handler classes.

To register a model, obtain an instance of *Registry* (this module exports one as *library*), and call its *register* method, passing the model class and a handler class (which should be a subclass of *Handler*). Note that both of these should be the actual classes, not instances of the classes.

To cease bookmarks handling for a model, call the *unregister* method, passing the model class.

For convenience, both *register* and *unregister* can also accept a list of model classes in place of a single model; this allows easier registration of multiple models with the same *Handler* class.

register(*self*, *model_or_iterable*, *handler_class=None*, ***kwargs*)

Register a model or a list of models for bookmark handling, using a particular *handler_class*, e.g.:

```
from bookmarks.handlers import library, Handler
# register one model
library.register(Article, Handler)
# register other two models
library.register([Film, Series], Handler)
```

If the handler class is not given, the default *bookmarks.handlers.Handler* class will be used.

If *kwargs* are present, they are used to override the handler class attributes (using instance attributes), e.g.:

```
library.register(Article, Handler,
    can_remove_bookmarks=False, form_class=MyForm)
```

Raise *AlreadyHandled* if any of the models are already registered. “”“

unregister(*self*, *model_or_iterable*)

Remove a model or a list of models from the list of models that will be handled.

Raise *NotHandled* if any of the models are not currently registered.

get_handler(*self*, *model_or_instance*)

Return the handler for given model or model instance. Return None if model is not registered.

Forms reference

class bookmarks.forms.**BookmarkForm**(*forms.Form*)

Form class to handle bookmarks.

The bookmark is identified by *model*, *object_id* and *key*. The bookmark is added or removed based on the his existance.

You can customize the app giving a custom form class, following some rules:

- the form must provide the following fields:
  - model -> a string representation of app label and model name of the bookmarked object (e.g.: ‘auth.user’)

– object_id -> the bookmarked instance id

– key -> the bookmark key

• the form must define the following methods:

– bookmark_exists(self): return True if the current user has that instance with that key in his bookmarks

– instance(self): return the current instance to bookmark or None if the form data (content_type_id and object_id) is invalid

– save(self): add or remove a bookmark and return it

__init__ (*self*, *request*, *backend*, **args*, ***kwargs*)
Takes the current *request*, the bookmark's *backend* and all the normal Django form arguments.

clean (*self*)
Check if an instance with current *model* and *object_id* actually exists in the database, and validate only if the user is authenticated.

instance (*self*)
Return the bookmarked instance or None if the form is not valid.

This method validates the form.

bookmark_exists (*self*)
Return True if *self.instance* is bookmarked by the current user with the current key.

Raise ValueError if the form is not valid.

This method validates the form.

save (*self*)
Add or remove the bookmark and return it.

You must call this method only after form validation.

Backends reference

The function of backends is to handle bookmarks retreival and save. They take care of things like adding or removing a bookmark, and getting all bookmarks based on some filters.

Writing your own backend

The application ships with a Django model backend and a MongoDB backend, but you can add your own defining a class with the interface below and pointing `settings.GENERIC_BOOKMARKS_BACKEND` to the new customized one.

class `bookmarks.backends.`**BaseBackend**
Base bookmarks backend.

Users may want to change `settings.GENERIC_BOOKMARKS_BACKEND` and customize the backend implementing all the methods defined here.

get_model (*self*)
Must return the bookmark model (a Django model or anything you like). Instances of this model must have the following attributes:

• user (who made the bookmark, a Django user instance)

• key (the bookmark key, as string)

- content_type (a Django content_type instance)
- object_id (a pk for the bookmarked object)
- content_object (the bookmarked object as a Django model instance)
- created_at (the date when the bookmark is created)

add(*self*, *user*, *instance*, *key*)
Must create a bookmark for *instance* by *user* using *key*. Must return the created bookmark (as a *self.get_model()* instance). Must raise *exceptions.AlreadyExists* if the bookmark already exists.

remove(*self*, *user*, *instance*, *key*)
Must remove the bookmark identified by *user*, *instance* and *key*. Must return the removed bookmark (as a *self.get_model()* instance). Must raise *exceptions.DoesNotExist* if the bookmark does not exist.

remove_all_for(*self*, *instance*)
Must delete all the bookmarks related to given *instance*.

filter(*self*, ***kwargs*)
Must return all bookmarks corresponding to given *kwargs*.

The *kwargs* keys can be:

- user: Django user object or pk
- instance: a Django model instance
- content_type: a Django ContentType instance or pk
- model: a Django model
- key: the bookmark key to use
- reversed: reverse the order of results

The bookmarks must be an iterable (like a Django queryset) of *self.get_model()* instances.

The bookmarks must be ordered by creation date (*created_at*): if *reversed* is True the order must be descending.

get(*self*, *user*, *instance*, *key*)
Must return a bookmark added by *user* for *instance* using *key*. Must raise *exceptions.DoesNotExist* if the bookmark does not exist.

exists(*self*, *user*, *instance*, *key*)
Must return True if a bookmark given by *user* for *instance* using *key* exists, False otherwise.

Django

The default backend used if `settings.GENERIC_BOOKMARKS_BACKEND` is *None* is `ModelBackend`, that uses Django models to store bookmarks.

class `bookmarks.backends.`**ModelBackend**(*BaseBackend*)
Bookmarks backend based on Django models.

This is used by default if no other backend is specified.

MongoDB

In order to use the MongoDB backend you must change your settings file like:

```
GENERIC_BOOKMARKS_BACKEND = 'bookmarks.backends.MongoBackend'
GENERIC_BOOKMARKS_MONGODB = {"NAME": "bookmarks"}
```

and then install MongoEngine:

```
pip install mongoengine
```

See *Customization* for a more complete explanation of MongoDB settings.

class bookmarks.backends.**MongoBackend**(*BaseBackend*)
Bookmarks backend based on MongoDB.

Class based views

The application provides two generic class based views (only available if you are using Django >= 1.3).

BookmarksForView

class bookmarks.views.generic.**BookmarksForView**(*BookmarksMixin*, *DetailView*)
Can be used to retreive and display a list of bookmarks of a given object.

This class based view accepts all the parameters that can be passed to *django.views.generic.detail.DetailView*.

For example, you can add in your *urls.py* a view displaying all bookmarks of a single active article:

```
from bookmarks.views.generic import BookmarksForView

urlpatterns = patterns('',
    url(r'^(?P<slug>[-\w]+)/bookmarks/$', BookmarksForView.as_view(
        queryset=Article.objects.filter(is_active=True)),
        name="article_bookmarks"),
)
```

You can also manage bookmarks order (default is by date descending) and bookmarks keys, in order to retreive only bookmarks for a given key, e.g.:

```
from bookmarks.views.generic import BookmarksForView

urlpatterns = patterns('',
    url(r'^(?P<slug>[-\w]+)/bookmarks/$', BookmarksForView.as_view(
        model=Article, key='mykey', reversed_order=False),
        name="article_bookmarks"),
)
```

Two context variables will be present in the template:

- *object*: the bookmarked article
- *bookmarks*: all the bookmarks of that article

The default template suffix is `'_bookmarks'`, and so the template used in our example is `article_bookmarks.html`.

context_bookmarks_name
The name of context variable containing bookmarks. Default is *'bookmarks'*.

key
The bookmarks key to use for retreiving bookmarks. Default is *None*.

reversed_order
If True, bookmarks are ordered by creation date descending. Default is True.

get_context_bookmarks_name (*self*, *obj*)
Get the variable name to use for the bookmarks.

get_key (*self*, *obj*)
Get the key to use to retreive bookmarks. If the key is None, use all keys.

order_is_reversed (*self*, *obj*)
Return True to sort bookmarks by creation date descending.

get_bookmarks (*self*, *obj*, *key*, *is_reversed*)
Return a queryset of bookmarks of *obj*.

BookmarksByView

class bookmarks.views.generic.**BookmarksByView** (*BookmarksMixin*, *DetailView*)
Can be used to retreive and display a list of bookmarks saved by a given user.

This class based view accepts all the parameters that can be passed to *django.views.generic.detail.DetailView*, with an exception: it is not mandatory to specify the model or queryset used to retreive the user (*django.contrib.auth.models.User* model is used by default).

For example, you can add in your *urls.py* a view displaying all bookmarks by a single active user:

```
from bookmarks.views.generic import BookmarksByView

urlpatterns = patterns('',
    url(r'^(?P<pk>\d+)/bookmarks/$', BookmarksByView.as_view(
        queryset=User.objects.filter(is_active=True)),
        name="user_bookmarks"),
)
```

You can also manage bookmarks order (default is by date descending) and bookmarks keys, in order to retreive only bookmarks for a given key, e.g.:

```
from bookmarks.views.generic import BookmarksByView

urlpatterns = patterns('',
    url(r'^(?P<pk>\d+)/bookmarks/$', BookmarksByView.as_view(
        key='mykey', reversed_order=False),
        name="user_bookmarks"),
)
```

Two context variables will be present in the template:

- *object*: the user
- *bookmarks*: all the bookmarks saved by that user

The default template suffix is `'_bookmarks'`, and so the template used in our example is `user_bookmarks.html`.

context_bookmarks_name
The name of context variable containing bookmarks. Default is *'bookmarks'*.

key
The bookmarks key to use for retreiving bookmarks. Default is *None*.

reversed_order
If True, bookmarks are ordered by creation date descending. Default is True.

get_context_bookmarks_name (*self*, *obj*)
Get the variable name to use for the bookmarks.

get_key (*self*, *obj*)
Get the key to use to retreive bookmarks. If the key is None, use all keys.

order_is_reversed (*self*, *obj*)
Return True to sort bookmarks by creation date descending.

get_bookmarks (*self*, *obj*, *key*, *is_reversed*)
Return a queryset of bookmarks saved by *obj* user.

Models reference

Objects defined here are only used if you store bookmarks using default Django model backend.

Base models

class bookmarks.models.**Bookmark** (*models.Model*)
A user's bookmark for a content object.

This is only used if the current backend stores bookmarks in the database using Django models.

content_type
the bookmarked instance content type

object_id
the bookmarked instance id

content_object
the bookmarked instance

key
the bookmark key

user
the user who bookmarked the instance (as a fk to *django.contrib.auth.models.User*)

created_at
the bookmark creation datetime

objects
the manager used is *bookmarks.managers.BookmarksManager* (see below)

In bulk selections

bookmarks.models.**annotate_bookmarks** (*queryset_or_model*, *key*, *user*, *attr='is_bookmarked'*)
Annotate *queryset_or_model* with bookmarks, in order to retreive from the database all bookmark values in bulk.

The first argument *queryset_or_model* must be, of course, a queryset or a Django model object. The argument *key* is the bookmark key.

The bookmarks are filtered using given *user*.

A boolean is inserted in an attr named *attr* (default='is_bookmarked') of each object in the generated queryset.

Usage example:

```
for article in annotate_bookmarks(Article.objects.all(), 'favourite',
    myuser, attr='has_a_bookmark'):
    if article.has_a_bookmark:
        print u"User %s likes article %s" (myuser, article)
```

Abstract models

class bookmarks.models.**BookmarkedModel** (*models.Model*)

Mixin for bookmarkable models.

Models subclassing this abstract model gain a *bookmarks* attribute allowing accessto the reverse generic relation to the *bookmarks.models.Bookmark*.

Managers

class bookmarks.managers.**BookmarksManager** (*models.Manager*)

Manager used by *Bookmark* model.

get_for (*self*, *content_object*, *key*, ***kwargs*)

Return the instance related to *content_object* and matching *kwargs*. Return None if a bookmark is not found.

filter_for (*self*, *content_object_or_model*, ***kwargs*)

Return all the instances related to *content_object_or_model* and matching *kwargs*. The argument *content_object_or_model* can be both a model instance or a model class.

filter_with_contents (*self*, ***kwargs*)

Return all instances retreiving content objects in bulk in order to minimize db queries, e.g. to get all objects bookmarked by a user:

```
for bookmark in Bookmark.objects.filter_with_contents(user=myuser):
    bookmark.content_object # this does not hit the db
```

add (*self*, *user*, *content_object*, *key*)

Add a bookmark, given the user, the model instance and the key.

Raise a *Bookmark.AlreadyExists* exception if that kind of bookmark is present in the db.

remove (*self*, *user*, *content_object*, *key*)

Remove a bookmark, given the user, the model instance and the key.

Raise a *Bookmark.DoesNotExist* exception if that kind of bookmark is not present in the db.

remove_all_for (*self*, *content_object*)

Remove all bookmarks for the given model instance.

The application uses this whenever a bookmarkable model instance is deleted, in order to mantain the integrity of the bookmarks table.

CHAPTER 2

Indices and tables

- genindex
- search

Python Module Index

b

Index

Symbols

A

B

C

D

E

F

G

H

I

K

M

N

O

P

R

S

U