django-generic-flatblocks Documentation

Martin Mahner

Nov 24, 2019

Contents

Full documentation: https://docs.elephant.house/django-generic-flatblocks/

If you want to add tiny snippets of text to your site, manageable by the admin backend, you would use either django-chunks or django-flatblocks. However, both of them have one problem: you are limited to a predefined content field; a "text" field in chunks and a "title" and "text" field in flatblocks.

django-generic-flatblocks solves this problem as it knows nothing about the content itself. You *attach* your hand made content node (a simple model) where you can define any fields you want.

CHAPTER 1

Contents:

1.1 Installation

This package is available through the python package index, pypi. You can install the latest version by:

```
pip install django-generic-flatblocks
```

Add the module to your `INSTALLED_APPS` in your settings:

```
INSTALLED_APPS = (
    ...
    'django_generic_flatblocks',
    'django_generic_flatblocks.contrib.gblocks',  # Optional sample models
)
```

Make sure that `django.core.context_processors.request` was added to your `TEMPLATE` options:

```
TEMPLATES = [{
    'BACKEND': 'django.template.backends.django.DjangoTemplates',
    'OPTIONS': {
        'context_processors': [
            'django.template.context_processors.request',
            ...
```

(Optional) Define the url prefix to your contrib.admin installation in the setting `ADMIN_URL_PREFIX`. Most commonly this is `/admin/`. Beware the trailing slash.

Migrate the database schemas:

```
./manage.py migrate
```

See *Quickstart* for a quick demonstration or *Detailed usage* for a detailed integration.

1.1.1 Local Development

Install the package using pipenv:

```
$ cd django-generic-flatblocks
$ pipenv install --dev
$ pipenv run test
```

You can run the testsuite against a variety of Python and Django versions with tox:

```
$ cd django-generic-flatblocks
$ tox
```

1.2 Quickstart

You can join unlimited of slug-strings or context-variables to one slug. Most commonly you will do this if you need to use the users LANGUAGE_CODE in your slug, to have different content nodes for every language:

```
{% load generic_flatblocks %}
{% gblock "website","title",LANGUAGE_CODE for "gblocks.Title" %}
```

The slug can also be a context variable:

```
{% with "website_teaser" as my_slug %}
    {% gblock my_slug for "gblocks.Text" %}
{% endwith %}
```

You can render each generic block with a template of your choice:

```
{% gblock "website_urgent_notice" for "gblocks.Text" with "urgent.html" %}
```

You can pass an integer as slug. In this case, generic-flatblocks will fetch the model instance with the primary key you named in slug. Basically this is a {% include %} tag on model level:

```
{% gblock 1 for "auth.user" with "current_user.html" %}
```

You can store the related object directly in the context using the "into" argument. This is the quickest way to display any model. The "for" and "as" arguments are ignored:

```
{% gblock 1 for "auth.user" into "the_user_object" %}
<p>The first user is {{ the_user_object.username }}!</p>
{% if the_user_object_admin_url %}<a href="{{ the_user_object_admin_url }}">edit</a>{
↪% endif %}
```

Let's create an flatblock with a "as" argument. We publish this block at the end of this page in a variable called `FOOTER`:

```
{% gblock "footer" for "gblocks.Text" as "FOOTER" %}

{{ FOOTER }}
```

1.3 Detailed usage

First of all, in every template you want to use generic-flatblocks, load the templatetags library:

```
{% load generic_flatblocks %}
```

Then define a content node using the `gblock` templatetag:

```
{% gblock "unique_slug" for "applabel.modelname" with "render/with/template.html" as
↪"variable" %}
```

1.3.1 The arguments in detail:

"unique_slug" (required):

The slug argument defines under which *key* the content is stored in your database. You can define as many slugs as you want, just use a comma as separator. You can use context-variables as well. Examples:

```
"homepage headline" becomes "homepage_headline"
"homepage","headline" becomes "homepage_headline"
"homepage_title",LANGUAGE_CODE becomes "homepage_title_en" (depends on the users␣
↪locale code)
"user",user.pk becomes "user_1" (depends on the primary key of the currently logged␣
↪in user)
```

You can pass an *integer* as the slug. This will cause the templatetag to fetch the model named in *for* with the primary key you named in *slug*. Example:

```
{% gblock 1 for "auth.user" with "path/to/template.html" %}
```

This will fetch the auth.User with the primary key 1 and renders this model object with the template "path/to/template.html". In this case, the `generic_object` in `None`. Basically this is a `{% include %}` tag on model level. This can also be a context variable.

for "applabel.modelname" (required):

The *for* argument defines, what content-node (model) will be used to store and display the content. The format is *appname.modelname*. For some contributed content-nodes see *Contributed content nodes* below. This argument can be a context-variable.

with "template_path" (optional):

You can define a template that is used for rendering the content node. If you do not provide any template, the default template `<applabel>/<modelname>/flatblock.html` is used. This argument can be a context-variable.

In this template are all context-variables from the *parent* template available plus some extra variables:

- `object`: This variable is the model-instance for the generic block.
- `generic_object`: This variable is the model-instance for the generic content object itself. Mostly you don't need this.
- `admin_url`: A URL to the change view of the current object. This variable is `None` if the current user has no change permissions for the object.

as "variable name" (optional):

If you provide a variable name, the *rendered content node* is stored in it. Otherwise it's displayed directly. This argument can be a context-variable.

into "variable_name" (optional):

If you provide a variable name, the *related object* is stored in it. No template rendering is done. The *with* and the *as* arguments are ignored. This argument can be a context-variable.

After calling the gblock templatetag, you have the same variables available as in the *with* template:

- `variable_name`: This variable is the model-instance for the generic block.
- `variable_name` + `"_genric_object"`: This variable is the model-instance for the generic content object itself. Mostly you don't need this.
- `variable_name` + `"_admin_url"`: A URL to the change view of the current object. This variable is `None` if the current user has no change permissions for the object.

This is the quickest way to display any model instance or content-node directly without creating a template:

```
{% gblock 1 for "auth.User" into "theuser" %}
The first user is {{ theuser.username }}! (<a href="{{ theuser_admin_url }}">edit</a>)
```

would be rendered as:

```
The first user is johndoe! (<a href="/admin/auth/user/1/">edit</a>)
```

Note: If you have *settings.TEMPLATE_DEBUG* set to *True* and your related object does not exist, the templatetag will raise a ObjectNotFound exception. It will fail silently if you set *settings.TEMPLATE_DEBUG* to *False* and return an (empty, not saved) instance of the related model.

1.4 Create your own content node

A content node is a simple django-model. No quirks. If you want to use a title and a textfield as your content-node, define a new model `Entry` in your application `myproject`:

```
from django.db import models
from django.contrib import admin
from django.utils.encoding import python_2_unicode_compatible

@python_2_unicode_compatible
class Entry(models.Model):
    title = models.CharField(max_length=255, blank=True)
    content = models.TextField(blank=True)

    def __str__(self):
        return self.title

admin.site.register(Entry)
```

Important: django-generic-flatblocks creates an empty content-node upon first request, so make sure each field has either it's default value or allow `blank=True`. Don't forget to register your Model in the admin backend, if you want to edit it there.

Then create a template `myproject/entry/flatblock.html` in your template directory. This template is the default template to render the content node, if you do not provide a unique template for it (*with* argument).

In this template are all context-variables from the *parent* template available plus some extra variables. See arguments/with above for details.

A common template source for the content node would be:

```
<h1>{{ object.title }}</h1>
{{ object.content|safe }}

{% if admin_url %}<a href="{{ admin_url }}">edit this</a>{% endif %}
```

In your templates, create a new content node using the templatetag:

```
{% gblock "about_me" for "myproject.Entry" %}
```

For some pre defined nodes see *Contributed content nodes*

1.5 Contributed content nodes

django-generic-flatblocks comes with some very commonly used content-nodes. They are not installed by default. To do so, insert `django_generic_flatblocks.contrib.gblocks` to your `INSTALLED_APPS` in your settings and resync your database: `./manage.py syncdb`.

The contributed content nodes are:

- **gblocks.Title**: A CharField rendered as a <h2> Tag.
- **gblocks.Text**: A TextField rendered as html paragraphs. (This is what django-chunks provides)
- **gblocks.Image**: A ImageField rendered as <img> Tag.
- **gblocks.TitleAndText**: A CharField and a TextField. (This is what django-flatblocks provides)
- **gblocks.TitleTextAndImage**: A CharField, TextField and ImageField

So if you want to display a title and textfield, use this templatetag for example:

```
{% gblock "about_me" for "gblocks.TitleAndText" %}
```

1.6 Changelog

1.6.1 v1.3 (2019-03-16):

- Django 2.1 compatibility and tests.
- Python 3.7 compatibility and tests.
- Pipenv support.

- General code and package cleanup.

1.6.2 v1.2.1 (2018-02-18):

- Python backwards compatbility and coverage improvements.

1.6.3 v1.2 (2018-02-18):

- Django 2.0 compatibility and tests.

1.6.4 v1.1.1 (2017-04-30):

- Django 1.11 compatibility and tests.

1.6.5 v1.1 (2017-03-18):

- Django 1.10 compatibility and tests.
- Python 3.6 compatibility.
- *TEMPLATE_DEBUG* setting is no longer honored to raise individual errors, in favor of standard *DEBUG*.

1.6.6 v1.0 (2016-03-23):

- Code cleanup and update for Django 1.8+. Python3 Support. Better test integration. Better docs.

1.6.7 v0.9.1 (2010-03-22):

- Django 1.2 compatibility! Fixed a bug where tests did not pass under Django 1.2. Thanks to Brian Rosner for this.

1.6.8 v0.9 (2010-02-25):

- Fixed a bug where an integer was not allowed as a part of a slug.

1.6.9 v0.4 (2009-09-08):

- Added Danish translation.
- Added better documentation.
- Added unittests.
- If you fetch a not existing "primary key" object the templatetag will fail silently if settings.TEMPLATE_DEBUG is False.

1.6.10 v0.3.0 (2009-03-21):

- Added the *into* argument. You can now display any instance directly without creating and rendering a template.

1.6.11 v0.2.1 (2009-03-20):

- You can now pass a context variable with a integer to fetch a specific object.

1.6.12 v0.2.0 (2009-03-20):

- Added the ability to pass an integer as slug. This will cause that the templatetag fetches the specific *for* model with the primary key named in *slug*.

1.6.13 v0.1.2 (2009-03-20):

- Switched from distutils to setuptools. Fixed whitespace.

1.6.14 v0.1.1 (2009-03-15):

- Fixed wrong upload path of a contributed, generic block

1.6.15 v0.1 (2009-03-13):

- Initial release