djangocms-blog Documentation

Release 1.2.3

Iacopo Spalletti

Feb 11, 2023

Contents

CHAPTER 1

djangocms-blog

django CMS blog application - Support for multilingual posts, placeholders, social network meta tags and configurable apphooks.

Supported Django versions:

- Django 2.2, 3.0, 3.1

Supported django CMS versions:

- django CMS 3.7, 3.8+

> **Warning:** For Django<2.2, django CMS<3.7 versions support, use djangocms-blog 1.1x.

> **Warning:** Version 1.2 introduce a breaking change for customized `BLOG_PERMALINK_URLS`. Check the permalinks documentation for update information.

1.1 Installation

See installation documentation

1.2 Features

See features documentation for all the features details

- Support for django-app-enabler autoconfiguration.
- Placeholder content editing

- Frontend editing using django CMS frontend editor
- Multilingual support using django-parler
- Twitter cards, Open Graph and Google+ snippets meta tags
- Optional simpler TextField-based content editing
- Multisite (posts can be visible in one or more Django sites on the same project)
- Per-Apphook configuration
- Configurable permalinks
- Configurable django CMS menu
- Per-Apphook templates set
- Auto Apphook setup
- Django sitemap framework
- django CMS Wizard integration
- Haystack index
- Desktop notifications
- Liveblog

1.3 Known djangocms-blog websites

See DjangoPackages for an updated list https://www.djangopackages.com/packages/p/djangocms-blog/

1.4 Contents

1.4.1 Installation

django CMS blog assumes a **completely setup and working django CMS project**. See django CMS installation docs for reference.

If you are not familiar with django CMS you are **strongly encouraged** to read django CMS documentation before installing django CMS blog, as setting it up and adding blog content require to use django CMS features which are not described in this documentation.

django CMS docs:

- django CMS tutorial
- django CMS user guide
- django CMS videos

django-app-enabler support

django-app-enabler is supported.

You can either

- Installation & configuration: `python -mapp_enabler install djangocms-blog`

- Autoconfiguration: `python -mapp_enabler enable djangocms_blog`

You can further customise the blog configuration, you can start by checking:

- *Modify templates*
- *Enable haystack support*
- *Attach the blog to a page*
- *Further configuration*

Installation steps

Note: The steps in this section are applied automatically by `django-app-enabler`, if you use it.

- Install djangocms-blog:

```
pip install djangocms-blog
```

- Add `djangocms_blog` and its dependencies to INSTALLED_APPS:

```
INSTALLED_APPS = [
    ...
    'filer',
    'easy_thumbnails',
    'aldryn_apphooks_config',
    'parler',
    'taggit',
    'taggit_autosuggest',
    'meta',
    'sortedm2m',
    'djangocms_blog',
    ...
]
```

Note: The following are minimal defaults to get the blog running; they may not be suited for your deployment.

- Add the following settings to your project:

```
THUMBNAIL_PROCESSORS = (
    'easy_thumbnails.processors.colorspace',
    'easy_thumbnails.processors.autocrop',
    'filer.thumbnail_processors.scale_and_crop_with_subject_location',
    'easy_thumbnails.processors.filters',
)
META_SITE_PROTOCOL = 'https'  # set 'http' for non ssl enabled websites
META_USE_SITES = True
```

- For meta tags support enable the needed types:

```
META_USE_OG_PROPERTIES=True
META_USE_TWITTER_PROPERTIES=True
META_USE_GOOGLEPLUS_PROPERTIES=True # django-meta 1.x+
META_USE_SCHEMAORG_PROPERTIES=True  # django-meta 2.x+
```

- Configure parler according to your languages:

```
PARLER_LANGUAGES = {
    1: (
        {'code': 'en',},
        {'code': 'it',},
        {'code': 'fr',},
    ),
    'default': {
        'fallbacks': ['en', 'it', 'fr'],
    }
}
```

Note: Since parler 1.6 this can be skipped if the language configuration is the same as `CMS_LANGUAGES`.

- Add the following to your `urls.py`:

```
url(r'^taggit_autosuggest/', include('taggit_autosuggest.urls')),
```

- Apply the migrations:

```
python manage.py migrate
```

- Add the blog application (see *Attach the blog to a page* below).

Modify templates

For standard djangocms-blog templates to work to must ensure a `content` block is available in the django CMS template used by the page djangocms-blog is attached to.

For example, in case the page use the `base.html` template, you must ensure that something like the following is in the template:

```
...
{% block content %}
    {% placeholder "page_content" %}
{% endblock content %}
...
```

Alternative you can override then `djangocms_blog/base.html` and extend a different block

```
...
{% block my_block %}
<div class="app app-blog">
    {% block content_blog %}{% endblock %}
</div>
{% endblock my_block %}
...
```

Enable haystack support

If you want to enable haystack support:

- install djangocms-blog with:

```
pip install djangocms-blog[search]
```

- add `aldryn_search` to `INSTALLED_APPS`
- configure haystack according to aldryn-search docs and haystack docs.
- if not using `aldryn_search`, you can define your own `search_indexes.py` by skipping `aldryn_search` installation and writing your index for blog posts by following haystack documentation.

Attach the blog to a page

- To start your blog you need to use AppHooks from django CMS to add the blog to a django CMS page; this step is not required when using Auto setup:
  - Create a new django CMS page
  - Go to **Advanced settings** and select Blog from the **Application** selector and create an **Application configuration**;
  - Eventually customise the Application instance name;
  - Publish the page
  - Restart the project instance to properly load blog urls.

  Check the *Attaching blog to the home page* section to attach the blog on the website home page.

Warning: After adding the apphook to the page you **cannot** change the **Instance Namespace** field for the defined **AppHokConfig**; if you want to change it, create a new one with the correct namespace, go in the CMS page **Advanced settings** and switch to the new **Application configuration**

- Add and edit blog by creating them in the admin or using the toolbar, and the use the django CMS frontend editor. to edit the blog content:
  - Create a new blog entry in django admin backend or from the toolbar
  - Click on "view on site" button to view the post detail page
  - Edit the post via djangocms frontend by adding / editing plugins
  - Publish the blog post by flagging the "Publish" switch in the blog post admin

Note: by default djangocms-blog uses django CMS plugins for content, this means you will **not** have a text field in the blog post admin, but you will have to visit the frontend blog page (hit "Visit on site" button on the upper right corner) and add django CMS plugins on the frontend. Check the tutorial for more details.

Further configuration

As django CMS heavily relies on external applications to provide its features, you may also want to check the documentation of external packages.

Please, refer to each application documentation on details.

- django-cms (framework and content plugins): http://django-cms.readthedocs.io/en
- django-filer (image handling): https://django-filer.readthedocs.io
- django-meta (meta tag handling): https://github.com/nephila/django-meta#installation

- django-parler (multi language support): https://django-parler.readthedocs.io/en/latest/quickstart.html#configuration
- aldryn-search (content search): https://github.com/aldryn/aldryn-search#usage>
- haystack (content search): http://django-haystack.readthedocs.io/en/stable/

Auto setup

`djangocms_blog` can install and configue itself if it does not find any attached instance of itself. This feature is enable by default and will create:

- a `BlogConfig` with default values
- a `Blog` CMS page and will attach `djangocms_blog` instance to it
- a **home page** if no home is found.

All the items will be created in every language configured for the website and the pages will be published. If not using **aldryn-apphook-reload** or **django CMS 3.2** auto-reload middleware you are required to reload the project instance after this. This will only work for the current website as detected by `Site.objects.get_current()`.

The auto setup is execute once for each server start but it will skip any action if a `BlogConfig` instance is found.

1.4.2 Features

Attaching blog to the home page

Add blog apphook to the home page

- Go to the django CMS page admin: http://localhost:8000/admin/cms/page
- Edit the home page
- Go to **Advanced settings** and select Blog from the **Application** selector and create an **Application configuration**;
- Eventually customise the Application instance name;
- Publish the page
- Restart the project instance to properly load blog urls.

Amend configuration

Permalinks must be updated to avoid blog urlconf swallowing django CMS page patters.

To avoid this add the following settings to you project:

```
BLOG_AVAILABLE_PERMALINK_STYLES = (
    ('full_date', _('Full date')),
    ('short_date', _('Year /  Month')),
    ('category', _('Category')),
)
BLOG_PERMALINK_URLS = {
    "full_date": "<int:year>/<int:month>/<int:day>/<str:slug>/",
    "short_date: "<int:year>/<int:month>/<str:slug>/",
```

(continues on next page)

(continued from previous page)

```
    "category": "<str:category>/<str:slug>/",
}
```

Notice that the last permalink type is no longer present.

Then, pick any of the three remaining permalink types in the layout section of the apphooks config linked ot the home page (at http://yoursite.com/admin/djangocms_blog/blogconfig/).'

Warning: Version 1.2 introduce a breaking change as it drops `url` function in favour of `path`. If you have customized the urls as documented above you **must** update the custom urlconf to path-based patterns.

Provide a custom URLConf

It's possible to completely customize the urlconf by setting `BLOG_URLCONF` to the dotted path of the new urlconf.

Example:

```
BLOG_URLCONF = 'my_project.blog_urls'
```

The custom urlconf can be created by copying the existing urlconf in `djangocms_blog/urls.py`, saving it to a new file `my_project.blog_urls.py` and editing it according to the custom needs.

Configurable permalinks

Blog comes with four different styles of permalinks styles:

- Full date: `YYYY/MM/DD/SLUG`
- Year / Month: `YYYY/MM/SLUG`
- Category: `CATEGORY/SLUG`
- Just slug: `SLUG`

As all the styles are loaded in the urlconf, the latter two does not allow to have CMS pages beneath the page the blog is attached to. If you want to do this, you have to override the default urlconfs by setting something like the following in the project settings:

```
BLOG_PERMALINK_URLS = {
    "full_date": "<int:year>/<int:month>/<int:day>/<str:slug>/",
    "short_date": "<int:year>/<int:month>/<str:slug>/",
    "category": "<str:category>/<str:slug>/",
    "slug": "<str:slug>/",
}
```

And change `post/` with the desired prefix.

Warning: Version 1.2 introduce a breaking change as it drops `url` function in favour of `path`. If you have customized the urls as documented above you **must** update the custom urlconf to path-based patterns.

Templates

To ease the template customisations a `djangocms_blog/base.html` template is used by all the blog templates; the templates itself extends a `base.html` template; content is pulled in the `content` block. If you need to define a different base template, or if your base template does not defines a `content` block, copy in your template directory `djangocms_blog/base.html` and customise it according to your needs; the other application templates will use the newly created base template and will ignore the bundled one.

Templates set

You can provide a different set of templates for the whole `djangocms-blog` application by configuring the `Blog configs` accordingly.

This would require you to customize **all** the templates shipped in `djangocms_blog/templates/djangocms_blog`; the easiest way would be to copy the **content** of `djangocms_blog/templates/djangocms_blog` into another folder in the `templates` folder in our project (e.g., something like `cp -a djangocms_blog/templates/djangocms_blog/* /path/my/project/templates/my_blog`).

To use the new templates set, go to the `Blog configs` admin (something like `http://localhost:8000/en/admin/djangocms_blog/blogconfig/1/change`) and enter a directory name in the **Template prefix** field in the **Apphook configuration** admin (in the *Layout* section): it will be the root of your custom templates set; following the example above, you should enter `my_blog` as directory name.

For more instruction regarding template override, please read Django documentation: Overriding templates (for your version of Django).

Plugin Templates

You can have different layouts for each plugin (i.e.: `Latest Blog Articles`, `Author Blog Articles List` etc), by having multiple templates for each plugin. Default plugin templates are located in the `plugins` folder of the folder specified by the **Template prefix**; by default they are located in `templates/djangocms_blog`.

By defining the setting `BLOG_PLUGIN_TEMPLATE_FOLDERS` you can allow multiple sets of plugin templates allowing for different views per plugin instance. You could, for example, have a plugin displaying latest posts as a list, a table or in masonry style.

New templates have the same names as the standard templates in the `plugins` folder (e.g: `latest_entries.html`, `authors.html`, `tags.html`, `categories.html`, `archive.html`).

When using this feature you **must** provide **all** the templates for the available plugins.

To use this feature define `BLOG_PLUGIN_TEMPLATE_FOLDERS` as a list of available templates. Each item of this list itself is a list of the form `('[folder_name]', '[verbose name]')`.

Example:

```
BLOG_PLUGIN_TEMPLATE_FOLDERS = (
    ('plugins', _('Default template')),     # reads from templates/djangocms_blog/
↪plugins/
    ('timeline', _('Vertical timeline')),   # reads from templates/djangocms_blog/
↪vertical/
    ('masonry', _('Masonry style')),        # reads from templates/djangocms_blog/
↪masonry/
)
```

Once defined, the plugin admin interface will allow content managers to select which template the plugin will use.

Admin customization

As any other django model admin, `PostAdmin` can be customized by extending it adding a subclass in one of the project's applications.

```
admin.site.unregister(Post)
@admin.register(Post)
class CustomPostAdmin(PostAdmin):
    ...
    you_attributes_and_methods
```

Customizing the fieldsets

Due to the logic in *djangocms_blog.admin.PostAdmin.get_fieldsets()* method, it's advised to customize the fieldsets by overriding two private attributes *djangocms_blog.admin.PostAdmin._fieldsets* and *djangocms_blog.admin.PostAdmin._fieldset_extra_fields_position*.

_fieldsets

`_fieldsets` attribute follow the standard Django Admin fieldset structure. This is the primary source for customization.

You can freely rearrange and remove fields from this structure as you would to in a normal `fieldsets` attribute; the only caveat is that you must ensure consistency extra fields position set by `_fieldset_extra_fields_position`.

_fieldset_extra_fields_position

As some fields are managed by settings and apphook config, they are added to the final `fieldsets` by `PostAdmin.get_fieldsets`; you can customize their position (or hide them) by overriding `_fieldset_extra_fields_position` attribute.

The attribute is a dictionary containing the fields name as key, and by providing their position in the fieldsets as tuple.

Use a 2-item tuple if the field must be appended at the row level (e.g.: `(None, {"fields": ["field_a", "field_b"]})`) or a 3-item tuple if the field must be appended in a subgroup (e.g.: `(None, {"fields": ["field_a", ["field_b"[]})`.

Example

```
admin.site.unregister(Post)
@admin.register(Post)
class CustomPostAdmin(PostAdmin):
    ...
    _fieldsets = [
        (None, {"fields": ["title", "subtitle", "slug", "publish", "categories"]}),
        (
            _("Info"),
            {
                "fields": [["tags"], ["date_published", "date_published_end", "date_
↪featured"], "app_config", "enable_comments"],
```

(continues on next page)

(continued from previous page)

```
                "classes": ("collapse",),
            },
        ),
        (
            _("Images"),
            {"fields": [["main_image", "main_image_thumbnail", "main_image_full"]],
↪"classes": ("collapse",)},
        ),
        (_("SEO"), {"fields": [["meta_description", "meta_title", "meta_keywords"]],
↪"classes": ("collapse",)}),
    ]
    _fieldset_extra_fields_position = {
        "abstract": [0, 1],
        "post_text": [0, 1],
        "sites": [1, 1, 0],
        "author": [1, 1],
        "enable_liveblog": None,
        "related": [1, 1, 0],
    }
```

This example will result in:

- `"enable_liveblog"`: hidden even if enabled in settings
- `"sites"`: added in the same subgroup as `"tags"`
- `"author"`: appended after the `"enable_comments"` field
- `"app_config"` field moved to `"Info"` fieldset

Filter function

You can add / remove / filter fields at runtime by defining a method on you custom admin and proving its name in *BLOG_ADMIN_POST_FIELDSET_FILTER*.

Method must take the following arguments:

- `fsets`: current fieldsets dictionary
- `request`: current admin request
- `obj` (default: `None`): current post object (if available)

and it must return the modified fieldsets dictionary.

Function example:

```
def fieldset_filter_function(fsets, request, obj=None):
    if request.user.groups.filter(name='Editor').exists():
        fsets[1][1]['fields'][0].append('author')  # adding 'author' field if user is␣
↪Editor
    return fsets
```

Setup social metatags rendering

djangocms-blog implements django-meta and it come ready to provide a fairly complete social meta tags set.

Custom metatags are rendered on the blog post detail page only, while on the list page (which is a basically django CMS page) you can use djangocms-page-meta to render meta tags based on the django CMS page object.

In order to enable its rendering you must follow two steps:

- Enable `django-meta` settings in the project `settings.py`

```
META_SITE_PROTOCOL = 'https'  # set 'http' for non ssl enabled websites
META_USE_SITES = True
META_USE_OG_PROPERTIES=True
META_USE_TWITTER_PROPERTIES=True
META_USE_GOOGLEPLUS_PROPERTIES=True # django-meta 1.x+
META_USE_SCHEMAORG_PROPERTIES=True  # django-meta 2.x+
```

- Include `meta/meta.html` in the `head` tag of the template used to render `djangocms-blog`.

  a. The recommended way is to include in your project base templates:

```
<html>
<head>
   <title>{% block title %}{% page_attribute 'title' %}{% endblock title %}</
↪title>
   {% include "meta/meta.html" %}
   ...
```

  b. alternatively djangocms-blog base template provide a `meta` block you can place in your templates to only include `meta.html` for the blog posts:

```
<html>
<head>
   <title>{% block title %}{% page_attribute 'title' %}{% endblock title %}</
↪title>
   {% block meta %}{% endblock meta %}
   ...
```

  c. If you are also using `djangocms-page-meta` use this base template to make the two packages interoperable:

```
{% load page_meta_tags %}
{% page_meta request.current_page as page_meta %}
<html>
<head>
   <title>{% block title %}{% page_attribute 'title' %}{% endblock title %}</
↪title>
   {% block meta %}
   {% include 'djangocms_page_meta/meta.html' with meta=page_meta %}
   {% endblock meta %}
   ...
```

For complete social meta tags rendering, configure default properties (see `BLOG_FB`, `BLOG_TWITTER`, `BLOG_GPLUS`/`BLOG_SCHEMAORG` in *Configuration*) and apphook ones.

Social shares

`djangocms_blog` integrates well with options for social shares. One of the many options available is Shariff which was developed by a popular German computer magazine.

To allow readers to share articles on Facebook, Twitter, LinkedIn or just email, add the share buttons to your `post_detail.html` template just before `</article>`.

If you decide to use Shariff this requires a `<div>` to be added (see documentation of shariff).

See below for a template tag that loads all required configurations and javascript files. The `<div>` is then replaced by `{% shariff %}`:

```
from django.conf import settings
from django import template

register = template.Library()

@register.inclusion_tag('djangocms_blog/shariff.html', takes_context=True)
def shariff(context, title=None, services=None, orientation=None):
    context['orientation'] = orientation if orientation else 'horizontal'
    context['services'] = escape(services if services else
                                 settings.SHARIFF['services'])  # MUST be configured␣
↪in settings.py
    if title:
        context['short_message'] = settings.SHARIFF.get('prefix', '') + title +\
                    settings.SHARIFF.get('postfix', '')
    if 'mail-url' in settings.SHARIFF:
        context['mail_url'] = settings.SHARIFF['mail-url']
    return(context)
```

And in `templates/djangocms_blog/shariff.html` you need:

```
{% load static sekizai_tags %}
{% addtoblock 'js' %}<script src="{% static 'js/shariff.min.js' %}"></script>{%␣
↪endaddtoblock %}
{% addtoblock 'css' %}<link href="{% static 'css/shariff.min.css' %}" rel="stylesheet
↪">{% endaddtoblock %}
<div class="shariff" data-services="{{services}}" data-orientation="{{orientation}}"{
↪% if mail_url %} data-mail-url="{{mail_url}}"{% endif %}{% if short_message %} data-
↪title="{{short_message}}"{% endif %}></div>
```

The shariff files `js/shariff.min.js` and `css/shariff.min.css` will need to be added to your static files. Also, a little configuration in `settings.py` is needed, e.g.,

```
SHARIFF = {
    'services': '["twitter", "facebook", "googleplus", "linkedin", "xing", "mail"]',
    'mail-url': 'mailto:',                  # optional
    'prefix':   'Have you seen this: "',    # optional
    'postfix':  '"',                        # optional
}
```

Media plugins - Vlog / Podcast

Publising a vlog or a podcast require more introspection on the plugin contents than is generally available to django CMS plugins.

djangocms-blog provides a mixin for plugin models and templatetags to help when dealing with this use case.

For convenience, an additional `media` placeholder has been added to the `Post` model.

Note: djangocms-blog **only** provide a generic interface to introspect media plugins but it does not provide any plugin for any media platform as they would be very hard to maintain as the platforms changes. Examples provided here are working at the time of writing but they may require changes to work.

Base classes

class djangocms_blog.media.base.**MediaAttachmentPluginMixin**

Base class for media-enabled plugins.

Items that needs implementation in subclasses:

- media_id: property that provides the object id on the external platform
- media_url: property that provides the media public URL
- _media_autoconfiguration: configuration dictionary (see documentation for details)

_media_autoconfiguration = {'callable': None, 'main_url': '', 'params': [], 'thumb

Configuration dictionary. **All** the keys are required:

- `'params'`: one or more regular expressions to retrieve the media ID according to the provided `media_url`. It **must** contain a capturing group called `media_id` (see examples below).
- `'thumb_url'`: URL of the intermediate resolution media cover (depending on the plaform). It supports string formatting via `format` by providing the return json for the media according to the plugic specification.
- `'main_url'`: URL of the maximum resolution media cover (depending on the plaform). It supports string formatting via `%`-formatting by providing the return json for the media according to the plugic specification.
- `'callable'`: in case the above information are not recoverable from the object URL, provide here the name of a method on the plugin model instance taking the `media_id` as parameter and that builds the data required by `thumb_url`, `media_url` strings to build the correct cover urls.

get_main_image()

URL of the media cover at maximum resolution

Return type str

get_thumb_image()

URL of the media cover at intermediate resolution

Return type str

media_id

ID of the object on the remote media.

Return type str

media_params

Retrieves the media information.

Minimal keys returned:

- `media_id`: object id on the external platform
- `url`: full url to the public version of the media

In case the `'callable'` key in py:attr:‘ _media_autoconfiguration‘ is not `None`, it will be called instead (as method on the current model instance) to retrieve the information with any required logic.

Returns media information dictionary

Return type dict

media_url

Public URL of the object on the remote media.

As you will likely have a `URL` on the plugin model, it's usually enough to return that value, but you are free to implement any way to retrieve it.

Return type str

`djangocms_blog.templatetags.djangocms_blog.`**`media_images`**(*context*, *post*, *main=True*)

Extract images of the given size from all the *djangocms_blog.media.base.MediaAttachmentPluginMixin* plugins in the `media` placeholder of the provided post.

Support `djangocms-video poster` field in case the plugin does not implement `MediaAttachmentPluginMixin` API.

Usage:

Parameters

- **`context`** (*dict*) – template context
- **`post`** (*djangocms_blog.models.Post*) – post instance
- **`main`** (*bool*) – retrieve main image or thumbnail

Returns list of images urls

Return type list

`djangocms_blog.templatetags.djangocms_blog.`**`media_plugins`**(*context*, *post*)

Extract *djangocms_blog.media.base.MediaAttachmentPluginMixin* plugins from the `media` placeholder of the provided post.

They can be rendered with `render_plugin` templatetag:

Parameters

- **`context`** (*dict*) – template context
- **`post`** (*djangocms_blog.models.Post*) – post instance

Returns list of *djangocms_blog.media.base.MediaAttachmentPluginMixin* plugins

Return type List[*djangocms_blog.media.base.MediaAttachmentPluginMixin*]

How to build the media plugins

1. **Create a plugin model** Create a plugin model inheriting from `CMSPlugin` or a subclass of it and add *djangocms_blog.media.base.MediaAttachmentPluginMixin* mixin.
2. **Provide the media configuration** Populate *djangocms_blog.media.base.MediaAttachmentPluginMixin._media_autoconfiguration*.
3. **Implement required properties** Provide an implementation for the following properties:
   - *djangocms_blog.media.base.MediaAttachmentPluginMixin.media_url*
4. **Add any additional properties / method suitable for your case.** See `media_title` field in the `Vimeo` model below.

Example

Plugin model

This is a basic example for a Vimeo plugin.

As covers cannot be calculated from the video id, we must download the related json and extract information from there. This model use the `'callable'` parameter in *djangocms_blog.media.base.MediaAttachmentPluginMixin._media_autoconfiguration*

```
class Vimeo(MediaAttachmentPluginMixin, CMSPlugin):
    url = models.URLField('Video URL')

    _media_autoconfiguration = {
        'params': [
            re.compile('^https://vimeo.com/(?P<media_id>[-0-9]+)$'),
        ],
        'thumb_url': '%(thumb_url)s',
        'main_url': '%(main_url)s',
        'callable': 'vimeo_data',
    }

    def __str__(self):
        return self.url

    @property
    def media_id(self):
        try:
            return self.media_params['id']
        except KeyError:
            return None

    @property
    def media_title(self):
        try:
            return self.media_params['title']
        except KeyError:
            return None

    @property
    def media_url(self):
        return self.url

    def vimeo_data(self, media_id):
        response = requests.get(
            'https://vimeo.com/api/v2/video/%(media_id)s.json' % {'media_id': media_
↪id, }
        )
        json = response.json()
        data = {}
        if json:
            data = json[0]
            data.update(
                {
                    'main_url': data['thumbnail_large'],
                    'thumb_url': data['thumbnail_medium'],
                }
            )
        return data
```

Plugin class

Plugin class does not require any special code / configuration and can be setup as usual.

```
@plugin_pool.register_plugin
class VimeoPlugin(CMSPluginBase):
    model = Vimeo
    module = 'Media'
    name = 'Vimeo'
    render_template = 'media_app/vimeo.html'
```

How to display information in templates

The actual implementation may vary a lot according to your design. To ease retrieving the plugins, check *djangocms_blog.templatetags.djangocms_blog.media_images()* *djangocms_blog.templatetags.djangocms_blog.media_plugins()* which abstract away a lot of the django CMS logic to retrieve the plugins for a placeholder.

It's important to remember that at least in *some* templates, you must have the `media` placeholder rendered using `{% render_placeholder post.media %}` templatetag, otherwise you will not be able to add the plugins to the blog post.

Example

Media plugin

The media plugin requires the normal template to render the video according to the plugin fields:

```
{% if instance.media_id %}<iframe src="https://player.vimeo.com/video/{{ instance.
↪media_id }}?badge=0&autopause=0&player_id=0&app_id=2221" width="1920" height="1080"␣
↪frameborder="0" title="{{ instance.media_title }}" allow="autoplay; fullscreen"␣
↪allowfullscreen></iframe>{% endif %}
```

Blog posts list

A basic implementation is retrieving the covers associated to each media content via *djangocms_blog.templatetags.djangocms_blog.media_images()* and rendering each with a `<img>` tag:

```
...
{% media_images post as previews %}
<div class="blog-visual">
  {% for preview in previews %}<img src="{{ preview }}" />{% endfor %}
</div>
...
```

Blog posts detail

A basic implementation is rendering the media plugins as you would do with normal plugins:

```
...
{% if not post.main_image_id %}
    <div class="blog-visual">{% render_placeholder post.media %}</div>
{% else %}
...
```

djangocms-video support

`poster` attribute from `djangocms-video` is also supported.

`poster` is just a static fixed-size image you can set to a `djangocms-video` instance, but adding the plugin to the `media` placeholder allows to extract the image from the field and display along with the generated previews by seamlessly using `media_images`.

The rendering of the full content is of course fully supported.

Menu

`djangocms_blog` provides support for django CMS menu framework.

By default all the categories and posts are added to the menu, in a hierarchical structure.

It is possibile to configure per Apphook, whether the menu includes post and categories (the default), only categories, only posts or no item.

If "post and categories" or "only categories" are set, all the posts not associated with a category are not added to the menu.

Sitemap

`djangocms_blog` provides a sitemap for improved SEO indexing. Sitemap returns all the published posts in all the languages each post is available.

The changefreq and priority is configurable per-apphook (see `BLOG_SITEMAP_*` in Global settings).

To add the blog Sitemap, add the following code to the project `urls.py`:

```
from cms.sitemaps import CMSSitemap
from djangocms_blog.sitemaps import BlogSitemap


urlpatterns = patterns(
    '',
    ...
    url(r'^sitemap\.xml$', 'django.contrib.sitemaps.views.sitemap',
        {'sitemaps': {
            'cmspages': CMSSitemap, 'blog': BlogSitemap,
        }
    }),
)
```

Multisite

django CMS blog provides full support for multisite setups.

Basic multisite

To enabled basic multisite add `BLOG_MULTISITE = True` to the project settings.

Each blog post can be assigned to none, one or more sites: if no site is selected, then it's visible on all sites. All users with permission on the blog can manage all the blog posts, whichever the sites are.

Multisite permissions

Multisite permissions allow to restrict users to only manage the blog posts for the sites they are enabled to

To implement the multisite permissions API, you must add a `get_sites` method on the user model which returns a queryset of sites the user is allowed to add posts to.

Example:

```
class CustomUser(AbstractUser):
    sites = models.ManyToManyField('sites.Site')

    def get_sites(self):
        return self.sites
```

Related posts

To each post a number of (sortable) related posts can be attached.

The default template implementation shows them a the bottom of the post detail, but it can be customized.

Channels: Desktop notifications - Liveblog

djangocms-blog implements some channels related features:

- desktop notifications
- liveblog

For detailed information on channels setup, please refer to channels documentation.

> **Warning:** channels support works only on Django 2.2 and up

Notifications

`djangocms-blog` integrates django-knocker to provide real time desktop notifications.

To enable notifications:

- Install **django-knocker** and **channels<2.0**
- Add `channels` and `knocker` application to `INSTALLED_APPS` together with channels:

```
INSTALLED_APPS = [
    ...
    'channels',
    'knocker',
    ...
]
```

- Load the `knocker` routing into channels configuration:

```
ASGI_APPLICATION = 'myproject.routing.application'
CHANNEL_LAYERS={
    'default': {
        'BACKEND': 'channels_redis.core.RedisChannelLayer',
        'CONFIG': {
            'hosts': [os.environ.get('REDIS_URL', 'redis://localhost:6379
↪')],
        },
    },
}
```

- Add to `myproject.routing.application` the knocker routes:

```
from channels.auth import AuthMiddlewareStack
from channels.routing import ProtocolTypeRouter, URLRouter
from django.urls import path
from knocker.routing import channel_routing as knocker_routing

application = ProtocolTypeRouter({
    'websocket': AuthMiddlewareStack(
        URLRouter([
            path('knocker/', knocker_routing),
        ])
    ),
})
```

- Load `{% static "js/knocker.js" %}` and `{% static "js/reconnecting-websocket.min.js" %}` into the templates
- Add the following code:

```
<script type="text/javascript">
  var knocker_language = '{{ LANGUAGE_CODE }}';
  var knocker_url = '/notifications';  // Set this to the actual URL
</script>
```

  The value of `knocker_url` must match the path configured in `myproject.routing.channel_routing.py`.

- Enable notifications for each Apphook config level by checking the **Send notifications on post publish** and **Send notifications on post update** flags in blog configuration model.

Liveblog

Liveblog feature allows to display any content on a single post in realtime.

This is implemented by creating a group for each liveblogging enabled post and assigning the clients to each group whenever they visit a liveblog post.

Each liveblogged text is a specialized plugin (see *extend_liveblog* for information on how to customize the liveblog plugin).

Enabling liveblog

To enable liveblog features:

- Add `djangocms_blog.liveblog` application to `INSTALLED_APPS` together with channels:

```
INSTALLED_APPS = [
    ...
    'channels',
    'djangocms_blog.liveblog',
    ...
]
```

- It's advised to configure `CMS_PLACEHOLDER_CONF` to only allow `Liveblog` plugins in `Liveblog` placeholder, and remove them from other placeholders:

```
CMS_PLACEHOLDER_CONF = {
    None: {
        'excluded_plugins': ['LiveblogPlugin'],
    }
    ...
    'liveblog': {
         'plugins': ['LiveblogPlugin'],
    }
    ...
}
```

- Add channels routing configuration:

```
ASGI_APPLICATION = 'myproject.routing.application'
CHANNEL_LAYERS={
    'default': {
        'BACKEND': 'channels_redis.core.RedisChannelLayer',
        'CONFIG': {
            'hosts': [os.environ.get('REDIS_URL', 'redis://localhost:6379
↪')],
        },
    },
}
```

Note: Check channels documentation for more detailed information on `CHANNEL_LAYERS` setup.

- Add to `myproject.routing.channel_routing.py` the knocker routes:

```
from channels.auth import AuthMiddlewareStack
from channels.routing import ProtocolTypeRouter, URLRouter
from django.urls import path

from djangocms_blog.liveblog.routing import channel_routing as djangocms_
↪blog_routing

application = ProtocolTypeRouter({
```

(continues on next page)

(continued from previous page)

```
    'websocket': AuthMiddlewareStack(
        URLRouter([
            path('liveblog/', djangocms_blog_routing),
        ])
    ),
})
```

- If you overwrite the post detail template, add the following code where you want to show the liveblog content:

```
{% if view.liveblog_enabled %}
    {% include "liveblog/includes/post_detail.html" %}
{% endif %}
```

Liveblob and notifications can be activated at the same time by configuring each.

Using liveblog

To use liveblog:

- Tick the `enable liveblog` flag in the `Info` fieldset;
- Open the blog post detail page;
- Optionally add static content to the `post content` placeholder; the default template will show static content on top of liveblog content; you can override the template for different rendering;
- Add plugins to the `Liveblog` placeholder;
- Tick the `publish` flag on each `Liveblog` plugin to send it to clients in realtime.

Extending liveblog plugin

Liveblog support ships with a default liveblog plugin that provides a title, a body and a filer image.

To customize the appearance of the plugin override the `liveblog/plugins/liveblog.html` template. Both the real time and non realtime version of the plugin will be rendered accordingly.

If you need something different, you can create your own plugin inheriting from `LiveblogInterface` and calling the method `self._post_save()` in the save method, after the model has been saved.

In `models.py`:

```
class MyLiveblog(LiveblogInterface, CMSPlugin):
"""
Basic liveblog plugin model
"""
text = models.TextField(_('text'))

def save(self, *args, **kwargs):
    super(MyLiveblog, self).save(*args, **kwargs)
    self._post_save()
```

The plugin class does not require any special inheritance; in `cms_plugins.py`:

```
class MyLiveblogPlugin(CMSPluginBase):
    name = _('Liveblog item')
    model = MyLiveblog
plugin_pool.register_plugin(MyLiveblogPlugin)
```

While not required, for consistency between between realtime and non realtime rendering, use the `publish` field inherited from `LiveblogInterface` to hide the plugin content when the plugin is not published.

django CMS 3.2+ Wizard

django CMS 3.2+ provides a content creation wizard that allows to quickly created supported content types, such as blog posts.

For each configured Apphook, a content type is added to the wizard.

Some issues with multiple registrations raising django CMS `AlreadyRegisteredException` hae been reported; to handle these cases gracefully, the exception is swallowed when Django `DEBUG == True` avoiding breaking production websites. In these cases they wizard may not show up, but the rest will work as intended.

Wizard can create blog post content by filling the `Text` form field. You can control the text plugin used for content creation by editing two settings:

- `BLOG_WIZARD_CONTENT_PLUGIN`: name of the plugin to use (default: `TextPlugin`)
- `BLOG_WIZARD_CONTENT_PLUGIN_BODY`: name of the plugin field to add text to (default: `body`)

Warning: the plugin used must only have the text field required, all additional fields must be optional, otherwise the wizard will fail.

Post Extensions

Posts can be extended to attach arbitrary fields to a post instance.

E.g. one want's to have in a template a field or placeholder related to a post.

```
{{ post.extension.some_field }}
{% render_placeholder post.placeholder_extension.some_placeholder %}
```

Define the models in your models.py

```
from cms.models import CMSPlugin, PlaceholderField

from djangocms_blog.models import Post

class PostExtension(models.Model):
    post = models.ForeignKey(Post, on_delete=models.CASCADE, related_name='extension')
    some_field = models.CharField(max_length=10)

class PostPlaceholderExtension(models.Model):
    post = models.OneToOneField(Post, on_delete=models.CASCADE, related_name=
↪'placeholder_extension')
    some_placeholder = PlaceholderField('some_placeholder')
```

Define a inline in your admin.py

```
from .models import PostExtension

class PostExtensionInline(admin.TabularInline):
    model = PostExtension
    fields = ['some_field']
    classes = ['collapse']
    extra = 1
    can_delete = False
    verbose_name = "PostExtension"
    verbose_name_plural = "PostExtensions"
```

Register the extension in djangocms_blog

```
import djangocms_blog.admin as blog_admin
blog_admin.register_extension(PostExtensionInline)
blog_admin.register_extension(PostPlaceholderExtension)
```

After this the inline will be available in the Post add and Post change admin forms and a PostPlaceholderExtension instance will be automatically created when a post object is created.

Upgrading django CMS blog

`djangocms-blog` uses the `ThumbnailOption` model from `cmsplugin-filer` / `django-filer`.

`ThumbnailOption` model used to be in `cmsplugin-filer` up to version 1.0 included.

Since `cmsplugin-filer` 1.1 has been moved to `django-filer` (since version 1.2).

`djangocms-blog` introduced compatibility shims to support both combinations.

Since `djangocms-blog` 1.0 the compatibility has been dropped and `django-filer` 1.3 is required for `djangocms-blog` to work.

See below the migration path if you still use older versions of `cmsplugin-filer` / `django-filer`.

Upgrading djangocms-blog from 0.9.x to 1.0

No specific migration path is needed:

- upgrade djangocms-blog to 1.0: `pip install djangocms-blog>=1.0`
- remove `cmsplugin_filer` from `INSTALLED_APPS`
- run migrations: `python manage.py migrate`

Upgrading djangocms-blog from 0.8.x to 1.0

- migrate to `djangocms-blog 0.8.12` following instructions below
- upgrade djangocms-blog to 1.0: `pip install djangocms-blog>=1.0`
- remove `cmsplugin_filer` from `INSTALLED_APPS`
- run migrations: `python manage.py migrate`

Upgrading cmsplugin-filer from 1.0 to 1.1

Due to changes in `cmsplugin-filer` / `django-filer` which moved `ThumbnailOption` model from the former to the latter, `djangocms-blog` must be migrated as well.

Migrating cmsplugin-filer to 1.1 and djangocms-blog up to 0.8.4

If you have djangocms-blog up to 0.8.4 (included) installed or you are upgrading from a previous djangocms-blog version together with cmsplugin-filer upgrade, you can apply the migrations:

```
pip install cmsplugin-filer==1.1.3 django-filer==1.2.7 djangocms-blog==0.8.4
python manage.py migrate
```

Migrating cmsplugin-filer to 1.1 and djangocms-blog 0.8.5+

If you already a djangocms-blog 0.8.5+ up to 0.8.11, upgrade to 0.8.11, then you have to de-apply some blog migrations when doing the upgrade:

```
pip install djangocms-blog==0.8.11
python manage.py migrate djangocms_blog 0017 ## reverse for these migration is a noop
pip install cmsplugin-filer==1.1.3 django-filer==1.2.7
python manage.py migrate
```

After this step you can upgrade to 0.8.12:

```
pip install djangocms-blog==0.8.12
```

Note: de-apply migration **before** upgrading cmsplugin-filer. If running before upgrade, the backward migration won't alter anything on the database, and it will allow the code to migrate `ThumbnailOption` from cmsplugin-filer to filer

---

Note: If you upgrade in a Django 1.10 environment, be sure to upgrade both packages at the same time to allow correct migration dependencies to be evaluated.

1.4.3 Configuration

Django settings

List of settings that can be set in project django settings.

`djangocms_blog.settings.`**`BLOG_ABSTRACT_CKEDITOR = True`**
Configuration for the CKEditor of the abstract field.

See https://github.com/divio/djangocms-text-ckeditor/#customizing-htmlfield-editor for details.

`djangocms_blog.settings.`**`BLOG_ADMIN_POST_FIELDSET_FILTER = False`**
Callable function to change (add or filter) fields to fieldsets for admin post edit form.

See *Filter function* for more details.

djangocms_blog.settings.**BLOG_ARCHIVE_PLUGIN_NAME = 'Archive'**
Name of the plugin showing the blog archive index.

djangocms_blog.settings.**BLOG_AUTHOR_DEFAULT = True**
Use a default if not specified:

- `True`: the current user is set as the default author;
- `False`: no default author is set;
- any other string: the user with the provided username is used;

djangocms_blog.settings.**BLOG_AUTHOR_POSTS_LIST_PLUGIN_NAME = 'Author Blog Articles List'**
Name of the plugin showing the list of posts per authors.

djangocms_blog.settings.**BLOG_AUTHOR_POSTS_PLUGIN_NAME = 'Author Blog Articles'**
Name of the plugin showing the list of blog posts authors.

djangocms_blog.settings.**BLOG_AUTO_APP_TITLE = 'Blog'**
Title of the `BlogConfig` instance created by *Auto setup*.

djangocms_blog.settings.**BLOG_AUTO_BLOG_TITLE = 'Blog'**
Title of the blog page created by *Auto setup*.

djangocms_blog.settings.**BLOG_AUTO_HOME_TITLE = 'Home'**
Title of the home page created by *Auto setup*.

djangocms_blog.settings.**BLOG_AUTO_NAMESPACE = 'Blog'**
Namespace of the `BlogConfig` instance created by *Auto setup*.

djangocms_blog.settings.**BLOG_AUTO_SETUP = True**
Enable the blog *Auto setup* feature.

djangocms_blog.settings.**BLOG_AVAILABLE_PERMALINK_STYLES = (('full_date', 'Full date'), ('sl**
Choices of permalinks styles.

djangocms_blog.settings.**BLOG_CATEGORY_PLUGIN_NAME = 'Categories'**
Name of the plugin showing the list of blog categories.

djangocms_blog.settings.**BLOG_CURRENT_NAMESPACE = 'djangocms_post_current_config'**
Current post config identifier in request.

Name of the request attribute used in `djangocms_blog.cms_toolbars.BlogToolbar` to detect the current apphook namespace.

djangocms_blog.settings.**BLOG_CURRENT_POST_IDENTIFIER = 'djangocms_post_current'**
Current post identifier in request.

Name of the request attribute used in `djangocms_blog.cms_toolbars.BlogToolbar` to detect if request match a post detail.

djangocms_blog.settings.**BLOG_DEFAULT_OBJECT_NAME = 'Article'**
Default label for Blog item (used in django CMS Wizard).

djangocms_blog.settings.**BLOG_DEFAULT_PUBLISHED = False**
Default post published status.

djangocms_blog.settings.**BLOG_ENABLE_COMMENTS = True**
Whether to enable comments by default on posts

While `djangocms_blog` does not ship any comment system, this flag can be used to control the chosen comments framework.

djangocms_blog.settings.**BLOG_ENABLE_SEARCH = True**
Enable `aldryn-search` (i.e.: `django-haystack`) indexes.

djangocms_blog.settings.**BLOG_ENABLE_THROUGH_TOOLBAR_MENU = False**
Show djangocms-blog toolbar in any page, even when outside the blog apphooks.

djangocms_blog.settings.**BLOG_FB_APPID = ''**
Facebook Application ID.

Default from FB_APPID defined in django-meta settings.

djangocms_blog.settings.**BLOG_FB_AUTHOR = 'get_author_name'**
Facebook profile URL of the post author.

djangocms_blog.settings.**BLOG_FB_AUTHOR_URL = 'get_author_url'**
Facebook profile URL of the post author.

djangocms_blog.settings.**BLOG_FB_PROFILE_ID = ''**
Facebook profile ID of the post author.

Default from FB_PROFILE_ID defined in django-meta settings.

djangocms_blog.settings.**BLOG_FB_PUBLISHER = ''**
Facebook URL of the blog publisher.

Default from FB_PUBLISHER defined in django-meta settings.

djangocms_blog.settings.**BLOG_FB_TYPE = 'Article'**
Open Graph type for the post object.

djangocms_blog.settings.**BLOG_FB_TYPES = (('Article', 'Article'), ('Website', 'Website'))**
Choices of available blog types.

Available values are defined in to META_FB_TYPES defined in django-meta settings.

djangocms_blog.settings.**BLOG_FEED_CACHE_TIMEOUT = 3600**
Cache timeout for RSS feeds.

djangocms_blog.settings.**BLOG_FEED_INSTANT_ITEMS = 50**
Number of items in Instant Article feed.

djangocms_blog.settings.**BLOG_FEED_LATEST_ITEMS = 10**
Number of items in latest items feed.

djangocms_blog.settings.**BLOG_FEED_TAGS_ITEMS = 10**
Number of items in per tags feed.

djangocms_blog.settings.**BLOG_IMAGE_FULL_SIZE = {'crop': True, 'size': '640', 'upscale':**
Easy-thumbnail alias configuration for the post main image when shown on the post detail; it's a dictionary with size, crop and upscale keys.

djangocms_blog.settings.**BLOG_IMAGE_THUMBNAIL_SIZE = {'crop': True, 'size': '120x120', 'up**
Easy-thumbnail alias configuration for the post main image when shown on the post lists; it's a dictionary with size, crop and upscale keys.

djangocms_blog.settings.**BLOG_LATEST_ENTRIES_PLUGIN_NAME = 'Latest Blog Articles'**
Name of the plugin showing the list of latest posts.

djangocms_blog.settings.**BLOG_LATEST_ENTRIES_PLUGIN_NAME_CACHED = 'Latest Blog Articles - Ca**
Name of the plugin showing the list of latest posts (cached version).

djangocms_blog.settings.**BLOG_LATEST_POSTS = 5**
Default number of post in the **Latest post** plugin.

djangocms_blog.settings.**BLOG_LIVEBLOG_PLUGINS = ('LiveblogPlugin',)**
List of plugins implementing the *Liveblog* feature.

djangocms_blog.settings.**BLOG_MENU_EMPTY_CATEGORIES = True**
Flag to show / hide categories without posts attached from the menu.

djangocms_blog.settings.**BLOG_MENU_TYPES = (('complete', 'Categories and posts'), ('categor**
List of available menu types.

djangocms_blog.settings.**BLOG_META_DESCRIPTION_LENGTH = 320**
Maximum length for the Meta description field.

djangocms_blog.settings.**BLOG_META_TITLE_LENGTH = 70**
Maximum length for the Meta title field.

djangocms_blog.settings.**BLOG_MULTISITE = True**
Add support for multisite setup.

djangocms_blog.settings.**BLOG_PAGINATION = 10**
Number of post per page.

djangocms_blog.settings.**BLOG_PERMALINK_URLS = {'category': '<str:category>/<str:slug>/',**
URLConf corresponding to *BLOG_AVAILABLE_PERMALINK_STYLES*.

djangocms_blog.settings.**BLOG_PLUGIN_MODULE_NAME = 'Blog'**
Name of the djangocms-blog plugins module (group).

djangocms_blog.settings.**BLOG_PLUGIN_TEMPLATE_FOLDERS = (('plugins', 'Default template'),)**
(Sub-)folder from which the plugin templates are loaded.

The default folder is `plugins`.

See *Plugin Templates* for more details.

djangocms_blog.settings.**BLOG_POSTS_LIST_TRUNCWORDS_COUNT = 100**
Default number of words shown for abstract in the post list.

djangocms_blog.settings.**BLOG_POST_TEXT_CKEDITOR = True**
Configuration for the CKEditor of the post content field.

See https://github.com/divio/djangocms-text-ckeditor/#customizing-htmlfield-editor for details.

djangocms_blog.settings.**BLOG_SCHEMAORG_AUTHOR = 'get_author_schemaorg'**
Google+ account of the post author (deprecated).

djangocms_blog.settings.**BLOG_SCHEMAORG_TYPE = 'Blog'**
Schema.org type for the post object.

djangocms_blog.settings.**BLOG_SCHEMAORG_TYPES = (('Article', 'Article'), ('Blog', 'Blog'),**
Choices of available Schema.org types.

Default from `SCHEMAORG_TYPES` defined in django-meta settings.

djangocms_blog.settings.**BLOG_SITEMAP_CHANGEFREQ = (('always', 'always'), ('hourly', 'hourly**
List for available changefreqs for sitemap items.

djangocms_blog.settings.**BLOG_SITEMAP_CHANGEFREQ_DEFAULT = 'monthly'**
Default changefreq for sitemap items.

djangocms_blog.settings.**BLOG_SITEMAP_PRIORITY_DEFAULT = '0.5'**
Default priority for sitemap items.

djangocms_blog.settings.**BLOG_TAGS_PLUGIN_NAME = 'Tags'**
Name of the plugin showing the tag blog cloud.

djangocms_blog.settings.**BLOG_TWITTER_AUTHOR = 'get_author_twitter'**
Twitter account of the post author.

djangocms_blog.settings.**BLOG_TWITTER_SITE = ''**
Twitter account of the site.

Default from TWITTER_SITE defined in django-meta settings.

djangocms_blog.settings.**BLOG_TWITTER_TYPE = 'summary'**
Twitter Card type for the post object.

djangocms_blog.settings.**BLOG_TWITTER_TYPES = (('summary', 'Summary Card'), ('summary_large_**
Choices of available blog types for twitter.

Default from TWITTER_TYPES defined in django-meta settings.

djangocms_blog.settings.**BLOG_TYPE = 'Article'**
Generic type for the post object.

djangocms_blog.settings.**BLOG_TYPES = (('Article', 'Article'), ('Website', 'Website'))**
Choices of available blog types.

Available values are defined in to META_OBJECT_TYPES defined in django-meta settings.

djangocms_blog.settings.**BLOG_URLCONF = 'djangocms_blog.urls'**
Standard Apphook URLConf.

djangocms_blog.settings.**BLOG_USE_ABSTRACT = True**
Use an abstract field for the post.

If False no abstract field is available for posts.

djangocms_blog.settings.**BLOG_USE_FALLBACK_LANGUAGE_IN_URL = False**
When displaying URL, prefer URL in the fallback language if an article or category is not available in the current language.

djangocms_blog.settings.**BLOG_USE_PLACEHOLDER = True**
Post content is managed via placeholder

If False a HTMLField is provided instead.

djangocms_blog.settings.**BLOG_USE_RELATED = True**
Enable related posts to link one post to others.

djangocms_blog.settings.**BLOG_WIZARD_CONTENT_PLUGIN = 'TextPlugin'**
Name of the plugin created by wizard for the text content.

djangocms_blog.settings.**BLOG_WIZARD_CONTENT_PLUGIN_BODY = 'body'**
Name of the plugin field to add wizard text.

djangocms_blog.settings.**MENU_TYPES = (('complete', 'Categories and posts'), ('categories',**
Types of menu structure.

djangocms_blog.settings.**PERMALINKS = (('full_date', 'Full date'), ('short_date', 'Year / Mo**
Permalinks styles.

djangocms_blog.settings.**PERMALINKS_URLS = {'category': '<str:category>/<str:slug>/', 'full**
Permalinks urlconfs.

djangocms_blog.settings.**SITEMAP_CHANGEFREQ_LIST = (('always', 'always'), ('hourly', 'hourly**
List of changefreqs defined in sitemaps.

djangocms_blog.settings.**get_setting**(*name*)
Get setting value from django settings with fallback to globals defaults.

djangocms_blog.settings.**params = {'BLOG_ABSTRACT_CKEDITOR': True, 'BLOG_ADMIN_POST_FIELDSET**

Per-apphook settings

class djangocms_blog.cms_appconfig.**BlogConfigForm**(*app_container*, *data=None*, *files=None*, *fields=()*, *exclude=()*, **args*, ***kwargs*)

Settings that can be changed per-apphook.

Their default value is the same as the corresponding Django settings, but it can be customized per each apphook and changed at runtime.

app_title = 'Blog'
Free text title that can be used as title in templates

default_image_full = None
Default size of full images

default_image_thumbnail = None
Default size of thumbnail images

default_published = None
Post published by default (default: *DEFAULT_PUBLISHED*)

gplus_author = None
Schema.org author name abstract field (default: *SCHEMAORG_AUTHOR*)

gplus_type = None
Schema.org object type (default: *SCHEMAORG_TYPE*)

menu_empty_categories = None
Show empty categories in menu (default: *MENU_EMPTY_CATEGORIES*)

menu_structure = None
Menu structure (default: `MENU_TYPE_COMPLETE`, see *MENU_TYPES*)

object_name = 'Article'
Free text label for Blog items in django CMS Wizard

object_type = None
Object type (default: *TYPE*, see *TYPES*)

og_app_id = None
Facebook application ID (default: *FB_PROFILE_ID*)

og_author = None
Facebook author (default: *FB_AUTHOR*)

og_author_url = None
Facebook author URL (default: *FB_AUTHOR_URL*)

og_profile_id = None
Facebook profile ID (default: *FB_PROFILE_ID*)

og_publisher = None
Facebook page URL (default: *FB_PUBLISHER*)

og_type = None
Facebook type (default: *FB_TYPE*, see *FB_TYPES*)

paginate_by = None
When paginating list views, how many articles per page? (default: *PAGINATION*)

send_knock_create = None
Send notifications on post update. Require channels integration

send_knock_update = None
Send notifications on post update. Require channels integration

set_author = None
Set author by default (default: *AUTHOR_DEFAULT*)

sitemap_changefreq = None
Sitemap changefreq (default: *SITEMAP_CHANGEFREQ_DEFAULT*, see: *SITEMAP_CHANGEFREQ*)

sitemap_priority = None
Sitemap priority (default: *SITEMAP_PRIORITY_DEFAULT*)

template_prefix = None
Alternative directory to load the blog templates from (default: "")

twitter_author = None
Twitter author handle (default: *TWITTER_AUTHOR*)

twitter_site = None
Twitter site handle (default: *TWITTER_SITE*)

twitter_type = None
Twitter type field (default: *TWITTER_TYPE*)

url_patterns = None
Structure of permalinks (get_absolute_url); see *AVAILABLE_PERMALINK_STYLES*

use_abstract = None
Use abstract field (default: *USE_ABSTRACT*)

use_placeholder = None
Use placeholder and plugins for article body (default: *USE_PLACEHOLDER*)

use_related = None
Enable related posts (default: *USE_RELATED*)

1.4.4 API Documentation

djangocms_blog.admin

Admin classes

class djangocms_blog.admin.**BlogCategoryAdmin**(*model*, *admin_site*)

form
alias of *djangocms_blog.forms.CategoryAdminForm*

get_prepopulated_fields(*request*, *obj=None*)
Hook for specifying custom prepopulated fields.

class djangocms_blog.admin.**BlogConfigAdmin**(*model*, *admin_site*)

get_fieldsets(*request*, *obj=None*)
Fieldsets configuration

save_model(*request*, *obj*, *form*, *change*)
Clear menu cache when changing menu structure

class djangocms_blog.admin.**PostAdmin**(*model*, *admin_site*)

_fieldset_extra_fields_position = {'abstract': (0, 1), 'author': (1, 1, 0), 'enable_
Indexes where to append extra fields.

Key: Supported extra / optional field name Value: None / 2 / 3 item tuple. If you want to hide the field in any case set `None` as dictionary value, otherwise use a tuple containing the index of the the "fields" attribute of the selected fieldset row and an optional third value if the target "fields" has subgroups.

_fieldsets = [(None, {'fields': ['title', 'subtitle', 'slug', 'publish', ['categories
Default fieldsets structure.

Follow the normal Django fieldsets syntax.

When customizing the structure, check the *_fieldset_extra_fields_position* to ensure extra fields position matches.

_get_extra_field_position(*field*)
Return the position in the fieldset where to add the given field.

_patch_fieldsets(*fsets*, *field*)
Patch the fieldsets list with additional fields, based on *_fieldset_extra_fields_position*.

_set_config_defaults(*request*, *form*, *obj=None*)
Cycle through app_config_values and sets the form value according to the options in the current apphook config.

self.app_config_values is a dictionary containing config options as keys, form fields as values:

```
app_config_values = {
    'apphook_config': 'form_field',
    ...
}
```

Parameters

- **request** – request object
- **form** – model form for the current model
- **obj** – current object

Returns form with defaults set

disable_comments(*request*, *queryset*)
Bulk action to disable comments for selected posts. queryset must not be empty (ensured by django CMS).

disable_liveblog(*request*, *queryset*)
Bulk action to disable comments for selected posts. queryset must not be empty (ensured by django CMS).

enable_comments(*request*, *queryset*)
Bulk action to enable comments for selected posts. queryset must not be empty (ensured by django CMS).

enable_liveblog(*request*, *queryset*)
Bulk action to enable comments for selected posts. queryset must not be empty (ensured by django CMS).

form
alias of *djangocms_blog.forms.PostAdminForm*

get_fieldsets(*request*, *obj=None*)
Customize the fieldsets according to the app settings

Parameters

- **request** – request
- **obj** – post

Returns fieldsets configuration

get_list_filter(*request*)
Return a sequence containing the fields to be displayed as filters in the right sidebar of the changelist page.

get_prepopulated_fields(*request*, *obj=None*)
Hook for specifying custom prepopulated fields.

get_queryset(*request*)
Make sure the current language is selected.

get_restricted_sites(*request*)
The sites on which the user has permission on.

To return the permissions, the method check for the `get_sites` method on the user instance (e.g.: `return request.user.get_sites()`) which must return the queryset of enabled sites. If the attribute does not exists, the user is considered enabled for all the websites.

Parameters **request** – current request

Returns boolean or a queryset of available sites

get_urls()
Customize the modeladmin urls

has_restricted_sites(*request*)
Whether the current user has permission on one site only

Parameters **request** – current request

Returns boolean: user has permission on only one site

make_published(*request*, *queryset*)
Bulk action to mark selected posts as published. If the date_published field is empty the current time is saved as date_published. queryset must not be empty (ensured by django CMS).

make_unpublished(*request*, *queryset*)
Bulk action to mark selected posts as unpublished. queryset must not be empty (ensured by django CMS).

publish_post(*request*, *pk*)
Admin view to publish a single post

Parameters

- **request** – request
- **pk** – primary key of the post to publish

Returns Redirect to the post itself (if found) or fallback urls

save_model(*request*, *obj*, *form*, *change*)
Given a model instance save it to the database.

save_related(*request*, *form*, *formsets*, *change*)
Given the `HttpRequest`, the parent `ModelForm` instance, the list of inline formsets and a boolean value based on whether the parent is being added or changed, save the related objects to the database. Note that at this point save_form() and save_model() have already been called.

djangocms_blog.admin.**create_post_post_save**(*model*)
A wrapper for creating create_instance function for a specific model.

Admin forms

class djangocms_blog.forms.**CategoryAdminForm**(**args*, ***kwargs*)

class djangocms_blog.forms.**PostAdminForm**(**args*, ***kwargs*)

djangocms_blog.cms_menus

class djangocms_blog.cms_menus.**BlogCategoryMenu**(**args*, ***kwargs*)
Main menu class

Handles all types of blog menu

get_nodes(*request*)
Generates the nodelist

Parameters **request** –

Returns list of nodes

class djangocms_blog.cms_menus.**BlogNavModifier**(*renderer*)
This navigation modifier makes sure that when a particular blog post is viewed, a corresponding category is selected in menu

modify(*request*, *nodes*, *namespace*, *root_id*, *post_cut*, *breadcrumb*)
Actual modifier function :param request: request :param nodes: complete list of nodes :param namespace: Menu namespace :param root_id: eventual root_id :param post_cut: flag for modifier stage :param breadcrumb: flag for modifier stage :return: nodeslist

djangocms_blog.cms_menus.**clear_menu_cache**(***kwargs*)
Empty menu cache when saving categories

djangocms_blog.models

Models

class djangocms_blog.models.**Post**(**args*, ***kwargs*)
Blog post

exception DoesNotExist

exception MultipleObjectsReturned

get_author()
Return the author (user) objects

get_keywords()
Returns the list of keywords (as python list) :return: list

get_tags()
Returns the list of object tags as comma separated list

is_published
Checks wether the blog post is *really* published by checking publishing dates too

save(**args*, ***kwargs*)
Handle some auto configuration during save

save_translation(*translation*, **args*, ***kwargs*)
Handle some auto configuration during save

should_knock(*signal_type*, *created=False*)
Returns whether to emit knocks according to the post state

class djangocms_blog.models.**BlogCategory**(**args*, ***kwargs*)
Blog category

exception DoesNotExist

exception MultipleObjectsReturned

save(**args*, ***kwargs*)
Save the current instance. Override this in a subclass if you want to control the saving process.

The 'force_insert' and 'force_update' parameters can be used to insist that the "save" must be an SQL insert or update (or equivalent for non-SQL backends), respectively. Normally, they should not be set.

class djangocms_blog.models.**BlogMetaMixin**

get_full_url()
Return the url with protocol and domain url

get_meta_attribute(*param*)
Retrieves django-meta attributes from apphook config instance :param param: django-meta attribute passed as key

Managers

class djangocms_blog.managers.**GenericDateQuerySet**(**args*, ***kwargs*)

class djangocms_blog.managers.**GenericDateTaggedManager**

get_months(*queryset=None*, *current_site=True*)
Get months with aggregate count (how much posts is in the month). Results are ordered by date.

get_queryset(**args*, ***kwargs*)
Return a new QuerySet object. Subclasses can override this method to customize the behavior of the Manager.

queryset_class
alias of *GenericDateQuerySet*

djangocms_blog.cms_plugins

Plugin models

class djangocms_blog.models.**BasePostPlugin**(**args*, ***kwargs*)

optimize(*qs*)
Apply select_related / prefetch_related to optimize the view queries :param qs: queryset to optimize :return: optimized queryset

class djangocms_blog.models.**LatestPostsPlugin**(*id*, *path*, *depth*, *numchild*, *placeholder*, *parent*, *position*, *language*, *plugin_type*, *creation_date*, *changed_date*, *cmsplugin_ptr*, *app_config*, *current_site*, *template_folder*, *latest_posts*)

exception DoesNotExist

exception MultipleObjectsReturned

copy_relations(*oldinstance*)
Handle copying of any relations attached to this plugin. Custom plugins have to do this themselves!

class djangocms_blog.models.**AuthorEntriesPlugin**(*id*, *path*, *depth*, *numchild*, *placeholder*, *parent*, *position*, *language*, *plugin_type*, *creation_date*, *changed_date*, *cmsplugin_ptr*, *app_config*, *current_site*, *template_folder*, *latest_posts*)

exception DoesNotExist

exception MultipleObjectsReturned

copy_relations(*oldinstance*)
Handle copying of any relations attached to this plugin. Custom plugins have to do this themselves!

class djangocms_blog.models.**GenericBlogPlugin**(*id*, *path*, *depth*, *numchild*, *placeholder*, *parent*, *position*, *language*, *plugin_type*, *creation_date*, *changed_date*, *cmsplugin_ptr*, *app_config*, *current_site*, *template_folder*)

exception DoesNotExist

exception MultipleObjectsReturned

Plugin forms

class djangocms_blog.forms.**BlogPluginForm**(**args*, ***kwargs*)
Base plugin form to inject the list of configured template folders from BLOG_PLUGIN_TEMPLATE_FOLDERS.

class djangocms_blog.forms.**LatestEntriesForm**(**args*, ***kwargs*)
Custom forms for BlogLatestEntriesPlugin to properly load taggit-autosuggest.

class djangocms_blog.forms.**AuthorPostsForm**(**args*, ***kwargs*)
Custom form for BlogAuthorPostsPlugin to apply distinct to the list of authors in plugin changeform.

Plugin classes

class djangocms_blog.cms_plugins.**BlogArchivePlugin**(*model=None*, *admin_site=None*)
Render the list of months with available posts.

get_exclude(*request*, *obj=None*)
Exclude 'template_folder' field if BLOG_PLUGIN_TEMPLATE_FOLDERS contains one folder.

model
alias of *djangocms_blog.models.GenericBlogPlugin*

render(*context*, *instance*, *placeholder*)
Render the plugin.

class djangocms_blog.cms_plugins.**BlogAuthorPostsListPlugin**(*model=None*, *admin_site=None*)

Render the list of posts for each selected author.

get_fields(*request*, *obj=None*)
Return the fields available when editing the plugin.

'template_folder' field is added if BLOG_PLUGIN_TEMPLATE_FOLDERS contains multiple folders.

class djangocms_blog.cms_plugins.**BlogAuthorPostsPlugin**(*model=None*, *admin_site=None*)

Render the list of authors.

form
alias of *djangocms_blog.forms.AuthorPostsForm*

get_fields(*request*, *obj=None*)
Return the fields available when editing the plugin.

'template_folder' field is added if BLOG_PLUGIN_TEMPLATE_FOLDERS contains multiple folders.

model
alias of *djangocms_blog.models.AuthorEntriesPlugin*

render(*context*, *instance*, *placeholder*)
Render the plugin.

class djangocms_blog.cms_plugins.**BlogCategoryPlugin**(*model=None*, *admin_site=None*)

Render the list of post categories.

get_exclude(*request*, *obj=None*)
Exclude 'template_folder' field if BLOG_PLUGIN_TEMPLATE_FOLDERS contains one folder.

model
alias of *djangocms_blog.models.GenericBlogPlugin*

render(*context*, *instance*, *placeholder*)
Render the plugin.

class djangocms_blog.cms_plugins.**BlogLatestEntriesPlugin**(*model=None*, *admin_site=None*)

Return the latest published posts which bypasses cache, taking into account the user / toolbar state.

form
alias of *djangocms_blog.forms.LatestEntriesForm*

get_fields(*request*, *obj=None*)
Return the fields available when editing the plugin.

'template_folder' field is added if BLOG_PLUGIN_TEMPLATE_FOLDERS contains multiple folders.

model
alias of *djangocms_blog.models.LatestPostsPlugin*

render(*context*, *instance*, *placeholder*)
Render the plugin.

class djangocms_blog.cms_plugins.**BlogLatestEntriesPluginCached**(*model=None*, *admin_site=None*)

Return the latest published posts caching the result.

class djangocms_blog.cms_plugins.**BlogPlugin**(*model=None*, *admin_site=None*)

form
alias of *djangocms_blog.forms.BlogPluginForm*

get_render_template(*context*, *instance*, *placeholder*)
Select the template used to render the plugin.

Check the default folder as well as the folders provided to the apphook config.

class djangocms_blog.cms_plugins.**BlogTagsPlugin**(*model=None*, *admin_site=None*)
Render the list of post tags.

get_exclude(*request*, *obj=None*)
Exclude 'template_folder' field if BLOG_PLUGIN_TEMPLATE_FOLDERS contains one folder.

model
alias of *djangocms_blog.models.GenericBlogPlugin*

render(*context*, *instance*, *placeholder*)
Render the plugin.

djangocms_blog.views

class djangocms_blog.views.**AuthorEntriesView**(***kwargs*)

get_context_data(***kwargs*)
Get the context for this view.

get_queryset()
Return the list of items for this view.

The return value must be an iterable and may be an instance of *QuerySet* in which case *QuerySet* specific behavior will be enabled.

class djangocms_blog.views.**BaseBlogListView**

class djangocms_blog.views.**BaseBlogView**

get_view_url()
This method is used by the get_translated_url template tag.

By default, it uses the view_url_name to generate an URL. When the URL args and kwargs are translatable, override this function instead to generate the proper URL.

model
alias of *djangocms_blog.models.Post*

optimize(*qs*)
Apply select_related / prefetch_related to optimize the view queries :param qs: queryset to optimize :return: optimized queryset

class djangocms_blog.views.**CategoryEntriesView**(***kwargs*)

get_context_data(***kwargs*)
Get the context for this view.

get_queryset()
Return the list of items for this view.

The return value must be an iterable and may be an instance of *QuerySet* in which case *QuerySet* specific behavior will be enabled.

class djangocms_blog.views.**PostArchiveView**(***kwargs*)

get_context_data(***kwargs*)
Get the context for this view.

get_queryset()
Return the list of items for this view.

The return value must be an iterable and may be an instance of *QuerySet* in which case *QuerySet* specific behavior will be enabled.

class djangocms_blog.views.**PostDetailView**(***kwargs*)

get_context_data(***kwargs*)
Insert the single object into the context dict.

get_queryset()
Return the *QuerySet* that will be used to look up the object.

This method is called by the default implementation of get_object() and may not be called if get_object() is overridden.

get_template_names()
Return a list of template names to be used for the request. May not be called if render_to_response() is overridden. Return the following list:

- the value of `template_name` on the view (if provided)
- the contents of the `template_name_field` field on the object instance that the view is operating upon (if available)
- `<app_label>/<model_name><template_name_suffix>.html`

class djangocms_blog.views.**PostListView**(***kwargs*)

class djangocms_blog.views.**TaggedListView**(***kwargs*)

get_context_data(***kwargs*)
Get the context for this view.

get_queryset()
Return the list of items for this view.

The return value must be an iterable and may be an instance of *QuerySet* in which case *QuerySet* specific behavior will be enabled.

1.4.5 Development & community

django CMS Blog is an open-source project.

You don't need to be an expert developer to make a valuable contribution - all you need is a little knowledge, and a willingness to follow the contribution guidelines.

Nephila

django CMS Blog was created by Iacopo Spalletti at Nephila and is released under BSD License.

Nephila is an active supporter of django CMS and its community, and it maintains overall control of the django CMS Blog repository.

Standards & policies

django CMS Blog is a django CMS application, and shares much of django CMS's standards and policies.

These include:

- guidelines and policies for contributing to the project, including standards for code and documentation
- standards for managing the project's development
- a code of conduct for community activity

Please familiarise yourself with this documentation if you'd like to contribute to django CMS Blog.

1.4.6 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

Types of Contributions

Report Bugs

Report bugs at https://github.com/nephila/djangocms-blog/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

Write Documentation

djangocms-blog could always use more documentation, whether as part of the official djangocms-blog docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/nephila/djangocms-blog/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

Get Started!

Ready to contribute? Here's how to set up `djangocms-blog` for local development.

1. Fork the `djangocms-blog` repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/djangocms-blog.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv djangocms-blog
$ cd djangocms-blog/
$ pip install -r requirements-test.txt
$ pip install -e .
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ tox
```

To get tox, pip install it into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

Development tips

This project allows you to use pre-commit to ensure an easy compliance to the project code styles.

If you want to use it, install it globally (for example with `pip3 install --user precommit`, but check *installation instruction <https://pre-commit.com/#install>*. When first cloning the project ensure you install the git hooks by running `pre-commit install`.

From now on every commit will be checked against our code style.

Check also the available tox environments with `tox -l`: the ones not marked with a python version number are tools to help you work on the project buy checking / formatting code style, running docs etc.

Testing tips

You can test your project using any specific combination of python, django and django cms.

For example `tox -epy37-django30-cms37` runs the tests on python 3.7, Django 3.0 and django CMS 3.7.

As the project uses pytest as test runner, you can pass any pytest option by setting the `PYTEST_ARGS` environment variable, usually by prepending to the `tox` command. Example:

```
PYTEST_ARGS=" -s  tests/test_plugins.py::PluginTest -p no:warnings" tox -epy37-
↪django30-cms37
```

Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. Pull request must be named with the following naming scheme:

   `<type>/(<optional-task-type>-)<number>-description`

   See below for available types.

2. The pull request should include tests.

3. If the pull request adds functionality, the docs should be updated. Documentation must be added in `docs` directory, and must include usage information for the end user. In case of public API method, add extended docstrings with full parameters description and usage example.

4. Add a changes file in `changes` directory describing the contribution in one line. It will be added automatically to the history file upon release. File must be named as `<issue-number>.<type>` with type being:

   - `.feature`: For new features.
   - `.bugfix`: For bug fixes.
   - `.doc`: For documentation improvement.
   - `.removal`: For deprecation or removal of public API.
   - `.misc`: For general issues.

   Check towncrier documentation for more details.

5. The pull request should work for all python / django / django CMS versions declared in tox.ini. Check the CI and make sure that the tests pass for all supported versions.

Release a version

1. Update authors file
2. Merge `develop` on `master` branch
3. Bump release via task: `inv tag-release (major|minor|patch)`
4. Update changelog via towncrier: `towncrier --yes`

5. Commit changelog with `git commit --amend` to merge with bumpversion commit
6. Create tag `git tag <version>`
7. Push tag to github
8. Publish the release from the tags page
9. If pipeline succeeds, push `master`
10. Merge `master` back on `develop`
11. Bump developement version via task: `inv tag-dev -l (major|minor|patch)`
12. Push `develop`

1.4.7 History

1.2.3 (2020-12-21)

Features

- Improve django-app-enabler support (#660)
- Update translations
- Update django-app-enabler information

1.2.2 (2020-12-20)

Features

- Add support for Python 3.9 (#657)

Bugfixes

- Handle unicode chars in reverse of Post and Category models, using Django path() method instead of url() (#653)

Improved Documentation

- Update docs to clarify how to add content (#636)

1.2.1 (2020-11-16)

Bugfixes

- Fix admin urlconf not matching path syntax (#648)

1.2.0 (2020-11-14)

- BREAKING CHANGE: Replace url() function with path() (#645)

Features

- Add support for django CMS 3.8 / Django 3.1 (#643)
- Update to modern tooling and port to github actions (#564)
- Add option to allow fallback language in post/category URLs (#546)
- Move post slug to top of post admin form (#567)
- Add blog post extensions (#569)
- Replace deprecated function calls (#571)
- Improve tag selection UX/UI (#614)
- Improve admin customization (#620)
- Improve documentation (#622)
- Add rtd config file and refactor test / docs dependencies (#624)
- Replace G+ metadata with Schema.org (#637)
- Update pre-commit checks to latest version (#639)

Bugfixes

- Do not let site crash on menu if there is a namespace mismatch (#532)
- Improve setup documentation (#541)
- Fix GA on pull request (#575)
- Fix coveralls failure on pull request in GA (#577)
- Fix link in PR template (#581)
- Skip haystack index creation if aldryn-search is not installed but haystack is (#584)
- Skip towncrier on develop / master branch (#591)
- Fix missing template folders selection in plugins (#595)
- Clarify documentation regarding templates customization (#595)
- Fix extra migration when customizing BLOG_PLUGIN_TEMPLATE_FOLDERS (#597)
- Set default pytest configuration (#598)
- Add missing condition for (date_published) on blog_meta template (#603)
- Fix python version declaration in tox (#606)
- Fix creating text plugin in wizard (#607)
- Split linting GA to its own file (#608)
- Doc improvements for usage with djangocms-page-meta (#613)
- Update linting (#618)
- Fix building docs (#632)

Improved Documentation

- Improve documentation to enable social meta tags rendering (#613)

Misc

- #593

1.1.1 (2020-05-15)

- Fix channels support
- Avoid admin exception for related posts when config is None
- Fix error when overriding templates folder

1.1.0 (2020-05-04)

- Add support for django 3.0
- Add BlogAuthorPostsListPlugin to show posts per author
- Add experimental support for django-app-enabler
- Remove cmsplugin_filer from installation docs
- Set minimum django-parler version to 2.0
- Reduce the maximum length of slug fields to 752 characters
- Fix duplicated authors in BlogAuthorPostsPlugin
- Fix to respect current locale for category names
- Improve documentation for meta tags

1.0.0 (2019-11-04)

- Add support for django CMS 3.7
- Add support for Python 3.7
- Add image size meta for Facebook
- Add support for django-parler >= 2
- Move to django-app-helper
- Drop support for Django < 1.11
- Drop support for django CMS < 3.5
- Drop older compatibilities

0.9.11 (2019-08-06)

- Use menu_empty_categories config for BlogCategoryPlugin
- Purge menu cache when deleting a BlogConfig

0.9.10 (2019-07-02)

- Fixed allow_unicode kwarg for AutoSlugField
- Fixed sphinx conf isort
- Set category as requested or not depending on the permalink setting

0.9.9 (2019-04-05)

- Fixed issue with thumbnails not being preserved in admin form
- Pinned django-taggit version

0.9.8 (2019-01-13)

- Fixed test environment in Django 1.8, 1.9
- Added related posts to templates / documentation
- Added a fix for multiple error messages when slug is not unique

0.9.7 (2018-05-05)

- Fixed subtitle field not added to the admin

0.9.6 (2018-05-02)

- Fixed string representation when model has no language
- Added subtitle field

0.9.5 (2018-04-07)

- Fixed jquery path in Django 1.9+"Fix jquery path in Django 1.9+
- Added configurable blog abstract/text CKEditor

0.9.4 (2018-03-24)

- Fixed migration error from 0.8 to 0.9

0.9.3 (2018-03-12)

- Added dependency on lxml used in feeds
- Fixed warning on django CMS 3.5
- Fixed wizard in Django 1.11
- Updated translations

0.9.2 (2018-02-27)

- Fixed missing migration

0.9.1 (2018-02-22)

- Added Django 1.11 support

0.9.0 (2018-02-20)

- Added support for django CMS 3.4, 3.5
- Dropped support for Django<1.8, django CMS<3.2.
- Added liveblog application.
- Refactored plugin filters: by default only data for current site are now shown.
- Added global and per site posts count to BlogCategory.
- Added option to hide empty categories from menu.
- Added standalone documentation at https://djangocms-blog.readthedocs.io.
- Enabled cached version of BlogLatestEntriesPlugin.
- Added plugins templateset.
- Improved category admin to avoid circular relationships.
- Dropped strict dependency on aldryn-search, haystack. Install separately for search support.
- Improved admin filtering.
- Added featured date to post.
- Fixed issue with urls in sitemap if apphook is not published
- Moved template to easy_thumbnails_tags template tag. Require easy_thumbnails >= 2.4.1
- Made HTML description and title fields length configurable
- Added meta representation for CategoryEntriesView
- Generated valid slug in wizard if the given one is taken
- Fixed error in category filtering when loading the for via POST
- Returned 404 in AuthorEntriesView if author does not exists
- Returned 404 in CategoryEntriesView if category does not exists
- Generate valid slug in wizard if the given one is taken
- Limit categories / related in forms only to current lan

0.8.13 (2017-07-25)

- Dropped python 2.6 compatibility
- Fixed exceptions in __str__
- Fixed issue with duplicated categories in menu

0.8.12 (2017-03-11)

- Fixed migrations on Django 1.10

0.8.11 (2017-03-04)

- Fixed support for aldryn-apphooks-config 0.3.1

0.8.10 (2017-01-02)

- Fix error in get_absolute_url

0.8.9 (2016-10-25)

- Optimized querysets
- Fixed slug generation in wizard

0.8.8 (2016-09-04)

- Fixed issue with one migration
- Improved support for django CMS 3.4

0.8.7 (2016-08-25)

- Added support for django CMS 3.4
- Fixed issue with multisite support

0.8.6 (2016-08-03)

- Set the correct language during indexing

0.8.5 (2016-06-26)

- Fixed issues with ThumbnailOption migration under mysql.

0.8.4 (2016-06-22)

- Fixed issues with cmsplugin-filer 1.1.

0.8.3 (2016-06-21)

- Stricter filer dependency versioning.

0.8.2 (2016-06-12)

- Aldryn-only release. No code changes

0.8.1 (2016-06-11)

- Aldryn-only release. No code changes

0.8.0 (2016-06-05)

- Added django-knocker integration
- Changed the default value of date_published to null
- Cleared menu cache when changing menu layout in apphook config
- Fixed error with wizard multiple registration
- Made django CMS 3.2 the default version
- Fixed error with on_site filter
- Removed meta-mixin compatibility code
- Changed slug size to 255 chars
- Fixed pagination setting in list views
- Added API to set default sites if user has permission only for a subset of sites
- Added Aldryn integration

0.7.0 (2016-03-19)

- Make categories non required
- Fix tests with parler>=1.6
- Use all_languages_column to admin
- Add publish button
- Fix issues in migrations. Thanks @skirsdeda
- Fix selecting current menu item according to menu layout
- Fix some issues with haystack indexes
- Add support for moved ThumbnailOption
- Fix Django 1.9 issues
- Fix copy relations method in plugins
- Mitigate issue when apphook config can't be retrieved
- Mitigate issue when wizard double registration is triggered

0.6.3 (2015-12-22)

- Add BLOG_ADMIN_POST_FIELDSET_FILTER to filter admin fieldsets
- Ensure correct creation of full URL for canonical urls
- Move constants to settings
- Fix error when no config is found

0.6.2 (2015-11-16)

- Add app_config field to BlogLatestEntriesPlugin
- Fix __str__ plugins method
- Fix bug when selecting plugins template

0.6.1 (2015-10-31)

- Improve toolbar: add all languages for each post
- Improve toolbar: add per-apphook configurable changefreq, priority

0.6.0 (2015-10-30)

- Add support for django CMS 3.2 Wizard
- Add support for Apphook Config
- Add Haystack support
- Improved support for meta tags
- Improved admin
- LatestPostsPlugin tags field has been changed to a plain TaggableManager field. A migration is in place to move the data, but backup your data first.

0.5.0 (2015-08-09)

- Add support for Django 1.8
- Drop dependency on Django select2
- Code cleanups
- Enforce flake8 / isort checks
- Add categories menu
- Add option to disable the abstract

0.4.0 (2015-03-22)

- Fix Django 1.7 issues
- Fix dependencies on python 3 when using wheel packages
- Drop Django 1.5 support
- Fix various templates issues
- UX fixes in the admin

0.3.1 (2015-01-07)

- Fix page_name in template
- Set cascade to set null for post image and thumbnail options

0.3.0 (2015-01-04)

- Multisite support
- Configurable default author support
- Refactored settings
- Fix multilanguage issues
- Fix SEO fields length
- Post absolute url is generated from the title in any language if current is not available
- If djangocms-page-meta and djangocms-page-tags are installed, the relevant toolbar items are removed from the toolbar in the post detail view to avoid confusings page meta / tags with post ones
- Plugin API changed to filter out posts according to the request.
- Django 1.7 support
- Python 3.3 and 3.4 support

0.2.0 (2014-09-24)

- **INCOMPATIBLE CHANGE**: view names changed!
- Based on django parler 1.0
- Toolbar items contextual to the current page
- Add support for canonical URLs
- Add transifex support
- Add social tags via django-meta-mixin
- Per-post or site-wide comments enabling
- Simpler TextField-based content editing for simpler blogs
- Add support for custom user models

0.1.0 (2014-03-06)

- First experimental release

1.5 Indices and tables

- genindex
- search

Python Module Index

d

Index

Symbols

A

B

C

D

E

F

G

H

I

L

S

T

U