djangocms-blog Documentation

Release 1.1.1

Iacopo Spalletti

Jun 24, 2020

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 1.11, 2.2, 3.0

Supported django CMS versions:

- django CMS 3.5+

1.1 Installation

See installation documentation

1.2 Features

See features documentation for all the features details

- 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.

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',
    ...
]
```

Then apply migrations:

```
python manage.py migrate
```

If you want to enable haystack support, in addition to the above:

- 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.

To enable taggit filtering support in the admin install djangocms-blog with:

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

Minimal configuration

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 = 'http'
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
```

- 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')),
```

- 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.

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

External applications configuration

Dependency applications may need configuration to work properly.

Please, refer to each application documentation on details.

- django-cms: http://django-cms.readthedocs.io/en/release-3.4.x/how_to/install.html
- django-filer: https://django-filer.readthedocs.io
- django-meta: https://github.com/nephila/django-meta#installation
- django-meta-mixin: https://github.com/nephila/django-meta-mixin#installation
- django-parler: https://django-parler.readthedocs.io/en/latest/quickstart.html#configuration
- django-taggit-autosuggest: https://bitbucket.org/fabian/django-taggit-autosuggest
- aldryn-search: https://github.com/aldryn/aldryn-search#usage>
- haystack: 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

If you want to attach the blog to the home page you have to adapt settings a bit otherwise the "Just slug" permalink will swallow any CMS page you create.

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': r'^(?P<year>\d{4})/(?P<month>\d{1,2})/(?P<day>\d{1,2})/(?P<slug>\w[-
↪\w]*)/$',
    'short_date': r'^(?P<year>\d{4})/(?P<month>\d{1,2})/(?P<slug>\w[-\w]*)/$',
    'category': r'^(?P<category>\w[-\w]*)/(?P<slug>\w[-\w]*)/$',
}
```

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/).'

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': r'^(?P<year>\d{4})/(?P<month>\d{1,2})/(?P<day>\d{1,2})/(?P<slug>\w[-
↪\w]*)/$',
    'short_date': r'^(?P<year>\d{4})/(?P<month>\d{1,2})/(?P<slug>\w[-\w]*)/$',
```

(continues on next page)

(continued from previous page)

```
    'category': r'^post/(?P<category>\w[-\w]*)/(?P<slug>\w[-\w]*)/$',
    'slug': r'^post/(?P<slug>\w[-\w]*)/$',
}
```

And change `post/` with the desired prefix.

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

By using **Apphook configuration** you can define a different templates set. To use this feature provide a directory name in **Template prefix** field in the **Apphook configuration** admin (in *Layout* section): it will be the root of your custom templates set.

Plugin Templates

Plugin templates live in the `plugins` folder of the folder specified by the **Template prefix**, or by default `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.

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.

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.

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 Global Settings

- BLOG_IMAGE_THUMBNAIL_SIZE: Size of the main image when shown on the post lists; it's a dictionary with `size`, `crop` and `upscale` keys; (default: `{'size': '120x120', 'crop': True, 'upscale': False}`)
- BLOG_IMAGE_FULL_SIZE: Size of the main image when shown on the post detail; it's a dictionary with `size`, `crop` and `upscale` keys; (default: `{'size': '640x120', 'crop': True,'upscale': False}`)
- BLOG_PAGINATION: Number of post per page; (default: `10`)
- BLOG_MENU_EMPTY_CATEGORIES: Flag to show / hide categories without posts attached from the menu; (default: `True`)
- BLOG_LATEST_POSTS: Default number of post in the **Latest post** plugin; (default: `5`)

- BLOG_POSTS_LIST_TRUNCWORDS_COUNT: Default number of words shown for abstract in the post list; (default: `100`)
- BLOG_TYPE: Generic type for the post object; (default: `Article`)
- BLOG_TYPES: Choices of available blog types; (default: to `META_OBJECT_TYPES` defined in django-meta settings)
- BLOG_FB_TYPE: Open Graph type for the post object; (default: `Article`)
- BLOG_FB_TYPES: Choices of available blog types; (default: to `META_FB_TYPES` defined in django-meta settings)
- BLOG_FB_APPID: Facebook Application ID
- BLOG_FB_PROFILE_ID: Facebook profile ID of the post author
- BLOG_FB_PUBLISHER: Facebook URL of the blog publisher
- BLOG_FB_AUTHOR_URL: Facebook profile URL of the post author
- BLOG_FB_AUTHOR: Facebook profile URL of the post author
- BLOG_TWITTER_TYPE: Twitter Card type for the post object; (default: `Summary`)
- BLOG_TWITTER_TYPES: Choices of available blog types for twitter; (default: to `META_TWITTER_TYPES` defined in django-meta settings)
- BLOG_TWITTER_SITE: Twitter account of the site
- BLOG_TWITTER_AUTHOR: Twitter account of the post author
- BLOG_GPLUS_TYPE: Google+ Snippet type for the post object; (default: `Blog`)
- BLOG_GPLUS_TYPES: Choices of available blog types for twitter; (default: to `META_GPLUS_TYPES` defined in django-meta settings)
- BLOG_GPLUS_AUTHOR: Google+ account of the post author
- BLOG_ENABLE_COMMENTS: 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; (default: `True`)
- BLOG_USE_ABSTRACT: Use an abstract field for the post; if `False` no abstract field is available for every post; (default: `True`)
- BLOG_USE_PLACEHOLDER: Post content is managed via placeholder; if `False` a HTMLField is provided instead; (default: `True`)
- BLOG_USE_RELATED: Enable related posts to link one post to others; (default: `True`)
- BLOG_MULTISITE: Add support for multisite setup; (default: `True`)
- BLOG_AUTHOR_DEFAULT: Use a default if not specified; if set to `True` the current user is set as the default author, if set to `False` no default author is set, if set to a string the user with the provided username is used; (default: `True`)
- BLOG_DEFAULT_PUBLISHED: If posts are marked as published by default; (default: `False`)
- BLOG_ADMIN_POST_FIELDSET_FILTER: Callable function to change(add or filter) fields to fieldsets for admin post edit form; (default: `False`). Function example:

```
def fieldset_filter_function(fsets, request, obj=None):
    if request.user.groups.filter(name='Editor').exists():
```

(continues on next page)

(continued from previous page)

```
        fsets[1][1]['fields'][0].append('author')  # adding 'author' field if user is␣
↪Editor
    return fsets
```

- BLOG_AVAILABLE_PERMALINK_STYLES: Choices of permalinks styles;
- BLOG_PERMALINK_URLS: URLConf corresponding to BLOG_AVAILABLE_PERMALINK_STYLES;
- BLOG_URLCONF: Apphoo URLConf; (default: `'djangocms_blog.urls'`);
- BLOG_DEFAULT_OBJECT_NAME: Default name for Blog item (used in django CMS Wizard);
- BLOG_AUTO_SETUP: Enable the blog **Auto setup** feature; (default: `True`)
- BLOG_AUTO_HOME_TITLE: Title of the home page created by **Auto setup**; (default: `Home`)
- BLOG_AUTO_BLOG_TITLE: Title of the blog page created by **Auto setup**; (default: `Blog`)
- BLOG_AUTO_APP_TITLE: Title of the `BlogConfig` instance created by **Auto setup**; (default: `Blog`)
- BLOG_SITEMAP_PRIORITY_DEFAULT: Default priority for sitemap items; (default: `0.5`)
- BLOG_SITEMAP_CHANGEFREQ: List for available changefreqs for sitemap items; (default: **always**, **hourly**, **daily**, **weekly**, **monthly**, **yearly**, **never**)
- BLOG_SITEMAP_CHANGEFREQ_DEFAULT: Default changefreq for sitemap items; (default: `monthly`)
- BLOG_CURRENT_POST_IDENTIFIER: Current post identifier in request (default `djangocms_post_current`)
- BLOG_CURRENT_NAMESPACE: Current post config identifier in request (default: `djangocms_post_current_config`)
- BLOG_ENABLE_THROUGH_TOOLBAR_MENU: Is the toolbar menu throught whole all applications (default: `False`)
- BLOG_PLUGIN_MODULE_NAME: Blog plugin module name (default: `Blog`)
- BLOG_LATEST_ENTRIES_PLUGIN_NAME: Blog latest entries plugin name (default: `Latest Blog Articles`)
- BLOG_AUTHOR_POSTS_PLUGIN_NAME: Blog author posts plugin name (default: `Author Blog Articles`)
- BLOG_TAGS_PLUGIN_NAME: Blog tags plugin name (default: `Tags`)
- BLOG_CATEGORY_PLUGIN_NAME: Blog categories plugin name (default: `Categories`)
- BLOG_ARCHIVE_PLUGIN_NAME: Blog archive plugin name (default: `Archive`)
- BLOG_FEED_CACHE_TIMEOUT: Cache timeout for RSS feeds
- BLOG_FEED_INSTANT_ITEMS: Number of items in Instant Article feed
- BLOG_FEED_LATEST_ITEMS: Number of items in latest items feed
- BLOG_FEED_TAGS_ITEMS: Number of items in per tags feed
- BLOG_PLUGIN_TEMPLATE_FOLDERS: (Sub-)folder from which the plugin templates are loaded. The default folder is `plugins`. It goes into the `djangocms_blog` template folder (or, if set, the folder named in the app hook). This allows, e.g., different templates for showing a post list as tables, columns, ... New templates have the same names as the standard templates in the `plugins` folder (`latest_entries.html`, `authors.html`, `tags.html`, `categories.html`, `archive.html`). Default behavior corresponds to this setting being `( ("plugins", _("Default template") ).` To add new templates add to this setting, e.g., `('timeline', _('Vertical timeline') ).`

- BLOG_META_DESCRIPTION_LENGTH: Maximum length for the Meta description field (default: `320`)
- BLOG_META_TITLE_LENGTH: Maximum length for the Meta title field (default: `70`)
- BLOG_ABSTRACT_CKEDITOR: Configuration for the CKEditor of the abstract field (as per https://github.com/divio/djangocms-text-ckeditor/#customizing-htmlfield-editor)
- BLOG_POST_TEXT_CKEDITOR: Configuration for the CKEditor of the post content field

Read-only settings

- BLOG_MENU_TYPES: Available structures of the Blog menu; (default list **Posts and Categories**, **Categories only**, **Posts only**, **None**)
- BLOG_MENU_TYPE: Structure of the Blog menu; (default: `Posts and Categories`)

Per-Apphook settings

The following settings can be configured for each `Apphook config`: the settings above will be used as defaults.

- application title: Free text title that can be used as title in templates;
- object name: Free text label for Blog items in django CMS Wizard;
- Post published by default: Per-Apphook setting for BLOG_DEFAULT_PUBLISHED;
- Permalink structure: Per-Apphook setting for BLOG_AVAILABLE_PERMALINK_STYLES;
- Use placeholder and plugins for article body: Per-Apphook setting for BLOG_USE_PLACEHOLDER;
- Use abstract field: Per-Apphook setting for BLOG_USE_ABSTRACT;
- Enable related posts: Per-Apphook setting for BLOG_USE_RELATED;
- Set author: Per-Apphook setting for BLOG_AUTHOR_DEFAULT;
- Paginate sizePer-Apphook setting for BLOG_PAGINATION;
- Template prefix: Alternative directory to load the blog templates from;
- Menu structure: Per-Apphook setting for BLOG_MENU_TYPE
- Show empty categories in menu: Per-Apphook setting for BLOG_MENU_EMPTY_CATEGORIES
- Sitemap changefreq: Per-Apphook setting for BLOG_SITEMAP_CHANGEFREQ_DEFAULT
- Sitemap priority: Per-Apphook setting for BLOG_SITEMAP_PRIORITY_DEFAULT
- Object type: Per-Apphook setting for BLOG_TYPE
- Facebook type: Per-Apphook setting for BLOG_FB_TYPE
- Facebook application ID: Per-Apphook setting for BLOG_FB_APP_ID
- Facebook profile ID: Per-Apphook setting for BLOG_FB_PROFILE_ID
- Facebook page URL: Per-Apphook setting for BLOG_FB_PUBLISHER
- Facebook author URL: Per-Apphook setting for BLOG_AUTHOR_URL
- Facebook author: Per-Apphook setting for BLOG_AUTHOR
- Twitter type: Per-Apphook setting for BLOG_TWITTER_TYPE
- Twitter site handle: Per-Apphook setting for BLOG_TWITTER_SITE

- Twitter author handle: Per-Apphook setting for BLOG_TWITTER_AUTHOR
- Google+ type: Per-Apphook setting for BLOG_GPLUS_TYPE
- Google+ author name: Per-Apphook setting for BLOG_GPLUS_AUTHOR
- Send notifications on post publish: Send desktop notifications when a post is published
- Send notifications on post update: Send desktop notifications when a post is updated

1.4.4 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.

1.4.5 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.

Branching model

When planning a code cotnribution, these is the project branching model:

- new features goes to develop
- bugfixes for releases goes to release/Y.Z.x branches
- master is just a snapshot of the latest stable release and should not be targeted

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/
$ python setup.py develop
```

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, just pip install them 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.

Pull Request Guidelines

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

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4, 3.5 and 3.6. Check https://travis-ci.org/nephila/djangocms-blog/pull_requests and make sure that the tests pass for all supported Python versions.

Tips

To run a subset of tests:

```
$ python cms_helper.py test tests.test_views
```

1.4.6 History

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

D

G

M