django-page-cms Documentation

Release 2.0.11

Batiste Bieler

Aug 01, 2023

CONTENTS

Welcome on the documentation of the simple multilingual Gerbi CMS (package name: django-page-cms). You track the latest changes to the code base on the github project page. To get more information about this CMS and its feature go to the *Introduction section*.

For a quick install:

```
$ pip install django-page-cms[full]; gerbi --create mywebsite
```

For more complete installations instructions go to the *Installation section*. To get source code with git use:

```
$ git clone git://github.com/batiste/django-page-cms.git django-page-cms
```

Build this documentation from cloned repo:

```
$ pip install .[docs]
$ python setup.py build_sphinx
```

CHAPTER

ONE

TABLE OF CONTENT

1.1 Introduction

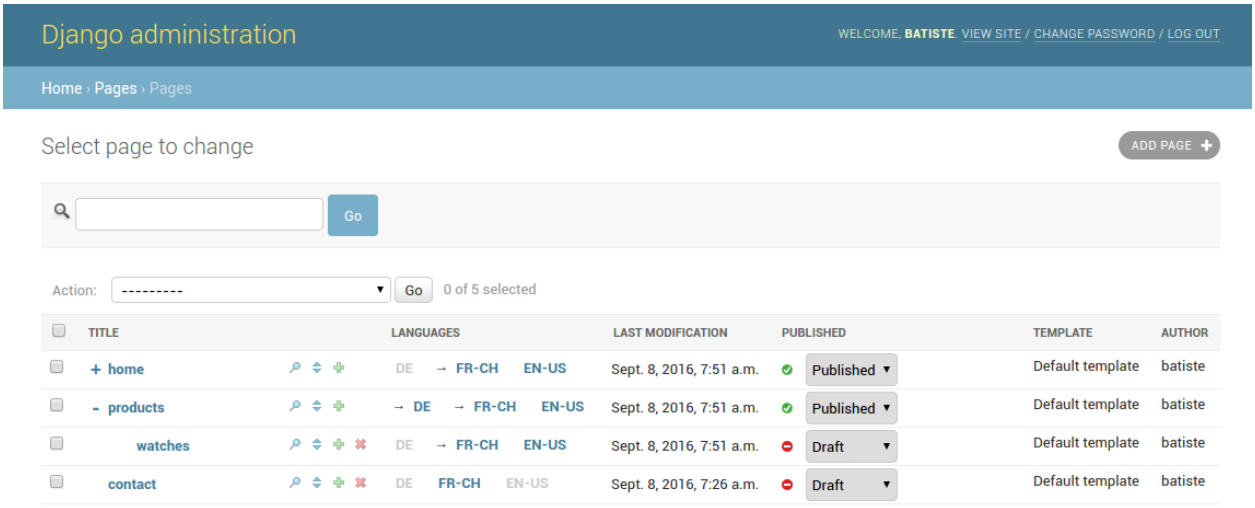
Gerbi CMS enable you to create and administrate hierarchical pages in a simple and powerful way.

Gerbi CMS is based around a placeholders concept. A placeholder is a template tag that you can use in your page's templates. Every time you add a placeholder in your template a field dynamically appears in the page admin.

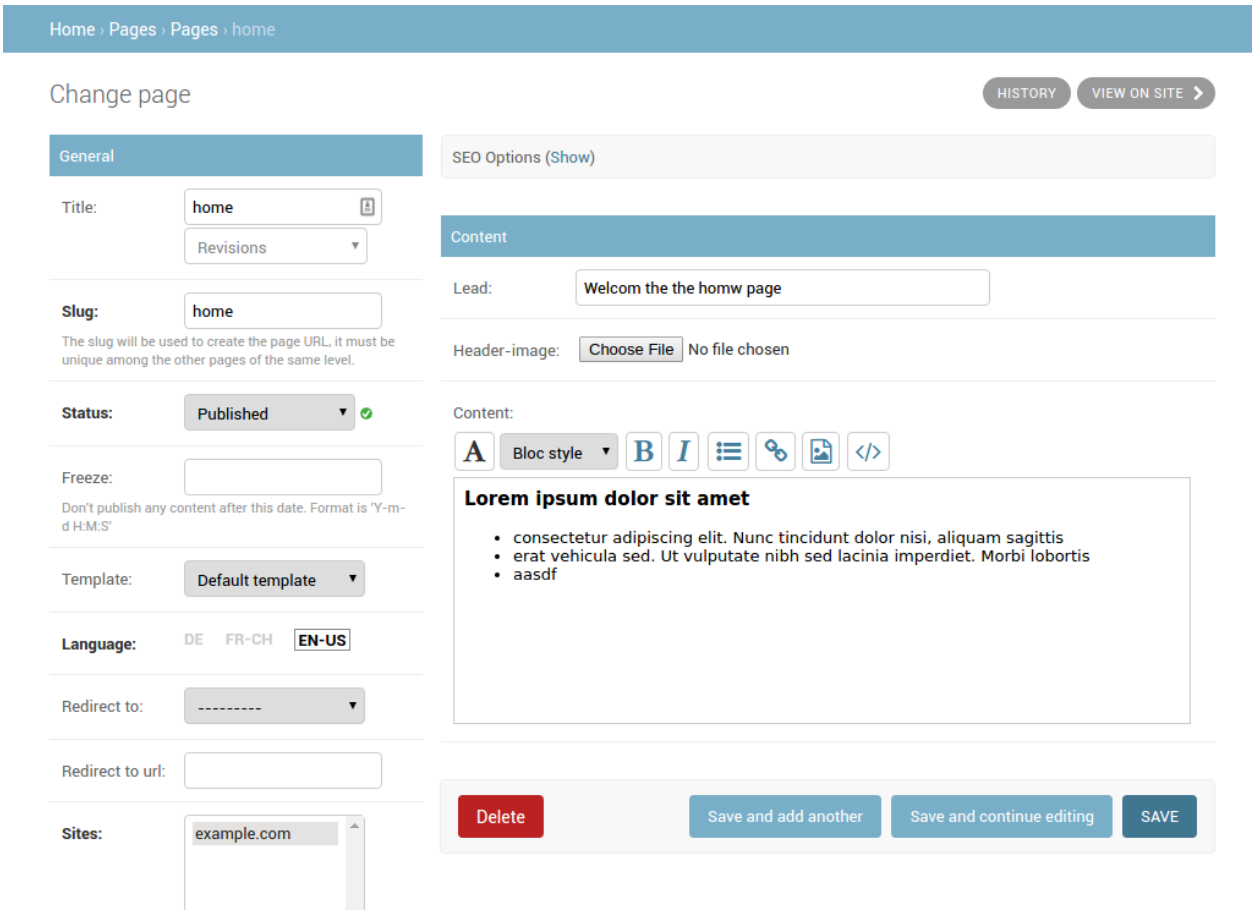
The project code repository is found at this address: http://github.com/batiste/django-page-cms

1.1.1 Screenshot

Admin page list



Admin page edition



1.1.2 Features

- *Automatic creation of localized placeholders* (content area) in admin by adding placeholders tags into page templates.
- Django admin application integration.
- Multilingual support.
- Inline editing.
- Media Library.
- Revisions.
- Plugin for page content export and import in JSON files within the admin interface
- Plugin for page content export and import in PO format for translations
- Search indexation with Django haystack.
- Advanced rights management (publisher, language manager).
- *Rich Text Editors* are directly available.
- Page can be moved in the tree in a visual way (drag & drop + simple click).
- The tree can be expanded/collapsed. A cookie remember your preferences.
- Possibility to specify a different page URL for each language.
- Directory-like page hierarchy (page can have the same name if they are not in the same directory).
- Every page can have multiple alias URLs. It's especially useful to migrate websites.
- *Possibility to integrate 3th party apps*.
- Image placeholder.
- Support for future publication start/end date.
- Page redirection to another page or arbirtrary URLs.
- Page tagging.
- Sites framework

1.1.3 Dependencies & Compatibility

- Django 2.0 or 3.0
- Python 3.6, 3.7, 3.8
- django-mptt is needed
- django-taggit if needed (if PAGE_TAGGING = True)
- django-haystack if needed
- Gerbi CMS is shipped with jQuery.
- Compatible with MySQL, PostgreSQL, SQLite3, some issues are known with Oracle.

Note: For install instruction go to the *Installation section*

1.1.4 How to contribute

Contributions section

1.1.5 Report a bug

Github issues

1.1.6 Internationalisation

This application is available in English, German, French, Spanish, Danish, Russian and Hebrew.

We use transifex

1.2 Contribute to Gerbi CMS

I recommend to create a fork on github and make modifications in your branch. Please follow those instructions:

- Add your name to the AUTHORS file.
- Write tests for any new code. Try to keep the test coverage >= 90%.
- Follow the PEP-08 as much as possible.
- If a new dependency is introduced, justify it.
- Be careful of performance regresssion.
- Every new CMS setting should start with PAGE_<something>
- Every new template_tag should start with pages_<something>

Then create a pull request. A short explanation of what you did and why you did it goes a long way.

1.2.1 Write tests

Gerbi CMS try to keep the code base stable. The test coverage is higher than 90% and we try to keep it that way.

To run all the tests:

```
$ python pages/test_runner.py
```

To run a specific test case:

```
$ python pages/test_runner.py pages.tests.test_selenium.SeleniumTestCase
```

To run a specific test in a test case:

```
$ python pages/test_runner.py pages.tests.test_selenium.SeleniumTestCase.test_admin_move_
↪page
```

1.2.2 Translations

We use transifex for translations

1.3 Placeholders template tag

Contents

The placeholder template tag is what make Gerbi CMS flexible. The workflow is that you design your template first according to the page design. Then you put placeholder tag where you want dynamic content to be. For each placeholder

added you will get a corresponding field in the administration interface so you can insert dynamic content. You can make as many templates as you want, and even use the template inheritance.

The syntax for a placeholder tag is the following:

```
{% placeholder "<name>" [on <page>] [with <widget>] [parsed] [inherited] [as <varname>]
↪%}
```

1.3.1 Enable the placeholder tags

The required syntax to load placeholder tags is the following:

```
{% load pages_tags %}
```

1.3.2 Detailed explanations on placeholder options

the on option

The **on** is provided to indicate the name of the page from which this placeholder should to get its content.

If the **on** option is omitted the CMS will automatically use the current page (by using the *current_page* context variable) to get the content of the placeholder.

Template syntax example

```
{% placeholder "main menu" on root_page %}
```

the with option

The **with** option is used to change a placeholder widget within the administration interface.

By default the CMS will use a simple *TextInput* widget. By using the **with** option you can change the widget to use. Widgets need to be registered before you can use them in the CMS:

```
from pages.widgets_registry import register_widget
from django.forms import TextInput

class NewWidget(TextInput):
    pass

register_widget(NewWidget)
```

Template syntax example:

```
{% placeholder "body" with NewWidget %}
```

Note: *Details about how to create a new Widget for a placeholder* .

Note: This CMS is shipped with *a list of useful widgets* .

The as option

If you use the option **as** the content of the placeholder content will not be displayed: a variable of your choice will be defined within the template's context.

Template syntax example:

```
{% placeholder "image" as image_src %}
<img src="{{ img_src }}" alt=""/>
```

The section option

The **section** option is used to group placeholders into a section in the admin interface. A section is collapsed by default and hides the fields.

Template syntax example:

```
<meta name="description" content="{% placeholder "meta_description" section "SEO" %}" />
<meta name="description" content="{% placeholder "meta_keywords" section "SEO" %}" />
```

You will get

The parsed keyword

If you add the keyword **parsed** the content of the placeholder will be evaluated as a Django template within the current context. Each placeholder with the **parsed** keyword will also have a note in the admin interface noting its ability to be evaluated as template.

Template syntax example:

```
{% placeholder "special content" parsed %}
```

The inherited keyword

If you add the keyword **inherited** the placeholder's content displayed on the frontend will be retrieved from the closest parent. But only if there is no content for the current page.

Template syntax example:

```
{% placeholder "right column" inherited %}
```

The untranslated keyword

If you add the keyword **untranslated** the placeholder's content will be the same whatever language you use. It's especialy useful for an image placeholder that should remain the same in every language.

Template syntax example:

```
{% imageplaceholder "logo" untranslated %}
```

The shared keyword

If you add the keyword **shared** the placeholder's content will be the same for all the pages. In effect the placeholder is not linked to any page and editing it change its content in all pages.

Template syntax example:

```
{% placeholder "footer-links" shared %}
```

The block keyword

Placeholders can also be rendered as template blocks. The content of the placeholder is then available as the *content* variable:

```
{% placeholder "title" with TextIntput block %}
    {% if content %}
        <h1>Welcome: {{ content }}</h1>
    {% endif %}
{% endplaceholder %}
```

Examples of other valid syntaxes

You can off course combine any of those syntaxes to your convenience. This is an example list of different possible syntaxes for this template tag:

```
{% placeholder "title" with TextIntput %}
{% placeholder "logo" untranslated on root_page %}
{% placeholder "right column" inherited as right_column parsed %}

...
<div class="my_funky_column">{{ right_column|safe }}</div>
```

1.3.3 Image placeholder

There is a special placeholder for images:

```
{% imageplaceholder "body image" block %}
    {% if content %}
        <img src="{{ MEDIA_URL }}{{ content }}" alt=""/>
    {% endif %}
{% endplaceholder %}
```

A file upload field will appears into the page admin interface.

1.3.4 File placeholder

There is also a more general placeholder for files:

```
{% fileplaceholder uploaded_file as filesrc block %}
    {% if content %}
        <a href="{{ MEDIA_URL }}{{ content }}">Download file</a>
    {% endif %}
{% endplaceholder %}
```

A file upload field will appears into the page admin interface.

1.3.5 Markdown placeholder

If you want to write in the MarkDown format there is a MarkDown placeholder:

```
{% markdownplaceholder mark %}
```

Note: You will have to install the Markdown library

```
$ pip install Markdown
```

1.3.6 Contact placeholder

If you want to include a simple contact form in your page, there is a contact placeholder:

```
{% contactplaceholder "contact" %}
```

This placeholder use *settings.ADMINS* for recipients email. The template used to render the contact form is *pages/contact.html*.

1.3.7 Create your own placeholder

If you want to create your own new type of placeholder, you can simply subclass the *PlaceholderNode*:

```
from pages.placeholders import PlaceholderNode
from pages.placeholders import parse_placeholder
register = template.Library()

class ContactFormPlaceholderNode(PlaceholderNode):

    def __init__(self, name, *args, **kwargs):
        ...

    def get_widget(self, page, language, fallback=Textarea):
        """Redefine this to change the widget of the field."""
        ...

    def get_field(self, page, language, initial=None):
        """Redefine this to change the field displayed in the admin."""
        ...

    def save(self, page, language, data, change):
        """Redefine this to change the way to save the placeholder data."""
        ...

    def render(self, context):
        """Output the content of the node in the template."""
        ...

def do_contactplaceholder(parser, token):
    name, params = parse_placeholder(parser, token)
    return ContactFormPlaceholderNode(name, **params)
register.tag('contactplaceholder', do_contactplaceholder)
```

And use it in your templates as a normal placeholder:

```
{% contactplaceholder contact %}
```

1.3.8 Changing the widget of the common placeholder

If you want to just redefine the widget of the default *PlaceholderNode* without subclassing it, you can just you create a valid Django Widget that take an extra language paramater:

```
from django.forms import Textarea
from django.utils.safestring import mark_safe
from pages.widgets_registry import register_widget

class CustomTextarea(Textarea):
    class Media:
        js = ['path to the widget extra javascript']
        css = {
            'all': ['path to the widget extra css']
```

(continues on next page)

(continued from previous page)

```
        }

    def __init__(self, language=None, attrs=None, **kwargs):
        attrs = {'class': 'custom-textarea'}
        super(CustomTextarea, self).__init__(attrs)

    def render(self, name, value, attrs=None):
        rendered = super(CustomTextarea, self).render(name, value, attrs)
        return mark_safe("""Take a look at \
                example.widgets.CustomTextarea<br>""") \
                + rendered

register_widget(CustomTextarea)
```

Create a file named *widgets.py* (or whathever you want) somewhere in one of your project's application. and then you can simply use the placeholder syntax:

```
{% placeholder custom_widget_example with CustomTextarea %}
```

Note: You have to make sure your *widgets.py* file is executed before using the widget. To be sure of this, you might import your file into the models.py of your application.

Note: More examples of custom widgets are available in *pages.widgets.py*.

1.3.9 List of placeholder widgets shipped with the CMS

Placeholder could be rendered with different widgets

TextInput

A simple line input:

```
{% placeholder "<name>" with TextInput %}
```

Textarea

A multi line input:

```
{% placeholder "<name>" with Textarea %}
```

AdminTextInput

A simple line input with Django admin CSS styling (better for larger input fields):

```
{% placeholder "<name>" with AdminTextInputWidget %}
```

AdminTextarea

A multi line input with Django admin CSS styling:

```
{% placeholder "<name>" with AdminTextareaWidget %}
```

FileBrowseInput

A file browsing widget:

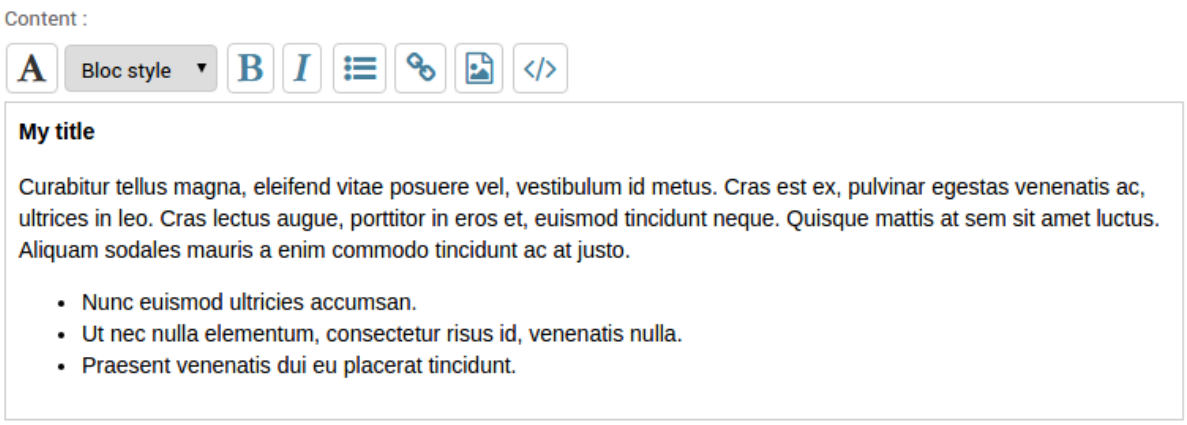
```
{% placeholder "<name>" with FileBrowseInput %}
```

Note: The following django application needs to be installed: https://github.com/sehmaschine/django-filebrowser

RichTextarea

A simple Rich Text Area Editor based on jQuery:

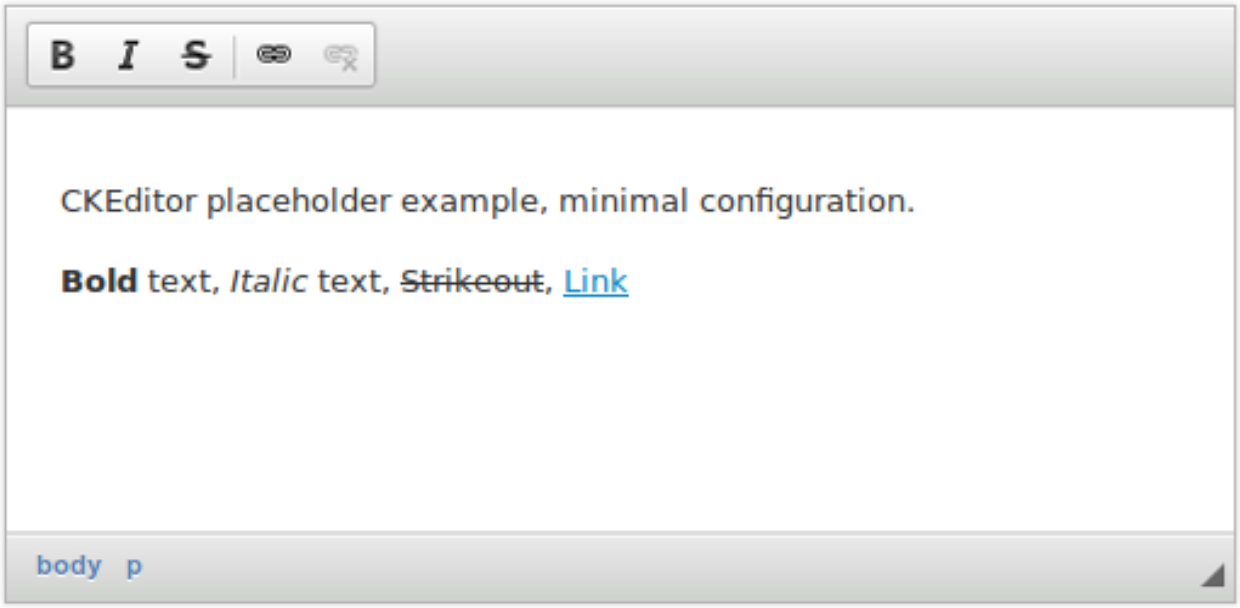
```
{% placeholder "<name>" with RichTextarea %}
```



CKEditorPlaceholderNode

A simple CKEditor custom placeholder:

```
{% ckeditor_placeholder "<name>" with ckeditor %}
```



The variable `CKEDITOR_CONFIGS` in `settings.py` exists to define editor configurations. To use a custom configured editor in templates, just specify the configuration name as follow:

```
{% ckeditor_placeholder "<name>" with ckeditor:minimal %}
```

If no configuration is specified a default configuration will be used. The default configuration has to be defined in `CKEDITOR_CONFIGS`. For example:

```
CKEDITOR_CONFIGS = {
    'default': {
        'width': 600,
        'height': 300,
        'toolbar': 'Full',
    }
}
```

Note: In order to use this placeholder, the application django-ckeditor needs to be installed and configured (trivial). See the docs for further details. Also have a look at the example project to see a working implementation.

1.4 Installation

This document explain how to install Gerbi CMS into an existing Django project. This document assume that you already know how to setup a Django project.

If you have any problem installing this CMS, take a look at the example application that stands in the example directory. This application works out of the box and will certainly help you to get started.

1.4.1 Evaluate quickly the application

Copy and paste this command in your virtual environnement and you should get a running cms instance:

```
$ pip install django-page-cms[full]; gerbi --create mywebsite
```

Then visit http://127.0.0.1:8000/

Or use docker:

```
$ docker-compose up
$ docker exec -it django-page-cms_web_1  python example/manage.py createsuperuser
$ docker exec -it django-page-cms_web_1  python example/manage.py pages_demo
```

1.4.2 Install dependencies by using pip

The pip install is the easiest and the recommended installation method. Use:

```
$ pip install django-page-cms[full]
```

1.4.3 Add django-page-cms into your INSTALLED_APPS

To activate django-page-cms you will need to add those application:

```
INSTALLED_APPS = (
    ...
    'mptt',
    'pages',
    ...
)
```

1.4.4 Urls

Take a look in the `example/urls.py` and copy desired URLs in your own `urls.py`. Basically you need to have something like this:

```
urlpatterns = patterns('',
    ...
    url(r'^pages/', include('pages.urls')),
    (r'^admin/', admin.site.urls),
)
```

When you will visit the site the first time (`/pages/`), you will get a 404 error because there is no published page. Go to the admin first and create and publish some pages.

1.4.5 Settings

All the Gerbi CMS specific settings and options are listed and explained in the `pages/settings.py` file.

Gerbi CMS require several of these settings to be set. They are marked in this document with a bold "*must*".

Note: If you want a complete list of the available settings for this CMS visit *the list of all available settings*.

Default template

You *must* set `PAGE_DEFAULT_TEMPLATE` to the path of your default CMS template:

```
PAGE_DEFAULT_TEMPLATE = 'pages/index.html'
```

This template must exist somewhere in your project. If you want you can copy the example templates from the directory `pages/templates/pages/examples/` into the directory `page` of your root template directory.

Additional templates

Optionally you can set `PAGE_TEMPLATES` if you want additional templates choices. In the the example application you have actually this:

```
PAGE_TEMPLATES = (
    ('pages/nice.html', 'nice one'),
    ('pages/cool.html', 'cool one'),
)
```

Media directory

The django CMS come with some javascript and CSS files. These files are standing in the `pages/static/pages` directory:

```
$ python manage.py collecstatic pages
```

And the cms media files will be copied in your project's media directory.

Languages

Please first read how django handle languages

- http://docs.djangoproject.com/en/dev/ref/settings/#languages
- http://docs.djangoproject.com/en/dev/ref/settings/#language-code

This CMS use the `PAGE_LANGUAGES` setting in order to present which language are supported by the CMS.

Django itself use the `LANGUAGES` setting to set the `request.LANGUAGE_CODE` value that is used by this CMS. So if the language you want to support is not present in the `LANGUAGES` setting the `request.LANGUAGE_CODE` will not be set correctly.

A possible solution is to redefine `settings.LANGUAGES`. For example you can do:

```
# Default language code for this installation. All choices can be found here:
# http://www.i18nguy.com/unicode/language-identifiers.html
LANGUAGE_CODE = 'en'

# This is defined here as a do-nothing function because we can't import
# django.utils.translation -- that module depends on the settings.
gettext_noop = lambda s: s

# here is all the languages supported by the CMS
PAGE_LANGUAGES = (
    ('de', gettext_noop('German')),
    ('fr', gettext_noop('Français')),
    ('en', gettext_noop('US English')),
)

# copy PAGE_LANGUAGES
languages = list(PAGE_LANGUAGES)

# redefine the LANGUAGES setting in order to be sure to have the correct request.
↪LANGUAGE_CODE
LANGUAGES = languages
```

Template context processors and Middlewares

You *must* have this context processors into your TEMPLATE_CONTEXT_PROCESSORS setting:

```
TEMPLATE_CONTEXT_PROCESSORS = (
    ...
    'pages.context_processors.media',
    ...
)
```

Caching

Gerbi CMS use the caching framework quite intensively. You should definitely setting-up a cache-backend to have decent performance.

If you want to setup a specific cache for Gerbi CMS instead of using the default you can do it by setting up the 'pages' cache entry:

```
CACHES = {
    'default': ...
    'pages': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': '127.0.0.1:11211',
    }
}
```

Note: The cache has been designed with memcache in mind: a single point of truth for cache. The CMS invalidates the cache actively when changes are made. That means that you need a central cache if you run this CMS in serveral processes otherwise the caches will become inconsitent.

The sites framework

If you want to use the Django sites framework with Gerbi CMS, you *must* define the SITE_ID and PAGE_USE_SITE_ID settings and create the appropriate Site object into the admin interface:

```
PAGE_USE_SITE_ID = True
SITE_ID = 1
```

The Site object should have the domain that match your actual domain (ie: 127.0.0.1:8000)

Tagging

Tagging is optional and disabled by default.

If you want to use it set PAGE_TAGGING at True into your setting file and add it to your installed apps:

```
INSTALLED_APPS = (
    ...
    'taggit',
    ...
)
```

1.5 Display page's content in templates

Gerbi CMS provide several template tags to extract data from the CMS. To use these tags in your templates you must load them first:

```
{% load pages_tags %}
```


1.5.1 get_content

Store a content type from a page into a context variable that you can reuse after:

```
{% get_content current_page "title" as content %}
```

You can also use the slug of a page:

```
{% get_content "my-page-slug" "title" as content %}
```

You can also use the id of a page:

```
{% get_content 10 "title" as content %}
```

Note: You can use either the page object, the slug, or the id of the page.

1.5.2 show_content

Output the content of a page directly within the template:

```
{% show_content current_page "title" %}
```

Note: You can use either the page object, the slug, or the id of the page.

1.5.3 page_has_content

Conditional tag that only renders its nodes if the page has content for a particular content type. By default the current page is used.

Syntax:

```
{% page_has_content <content_type> [<page var name>] %}
    ...
{% end page_has_content %}
```

Example:

```
{% page_has_content 'header-image' %}
    <img src="{{ MEDIA_URL }}{% imageplaceholder 'header-image' %}">
{% end_page_has_content %}
```

1.5.4 get_page

Retrieve a Page object and store it into a context variable that you can reuse after. Here is an example of the use of this template tag to display a list of news:

```
<h2>Latest news</h2>
{% get_page "news" as news_page %}
<ul>
{% for new in news_page.get_children %}
<li>
    <h3>{{ new.title }}</h3>
    {{ new.publication_date }}
    {% show_content new body %}
</li>
{% endfor %}
</ul>
```

Note: You can use either the slug, or the id of the page.

1.5.5 show_absolute_url

This tag show the absolute url of a page. The difference with the *Page.get_url_path* method is that the template knows which language is used within the context and display the URL accordingly:

```
{% show_absolute_url current_page %}
```

Note: You can use either the page object, the slug, or the id of the page.

1.5.6 Delegate the page rendering to another application

You can set another application to render certain pages of your website.

1.5.7 Subclass the default view

This CMS use a class based view. It is therefor easy to override the default behavior of this view. For example if you want to inject additional context information into all the pages templates you can override the method extra_context:

```
from pages.views import Details
from news.models import News

class NewsView(Details):

    def extra_context(self, request, context):
        lastest_news = News.object.all()
        context.update({'news': lastest_news})

details = NewsView()
```

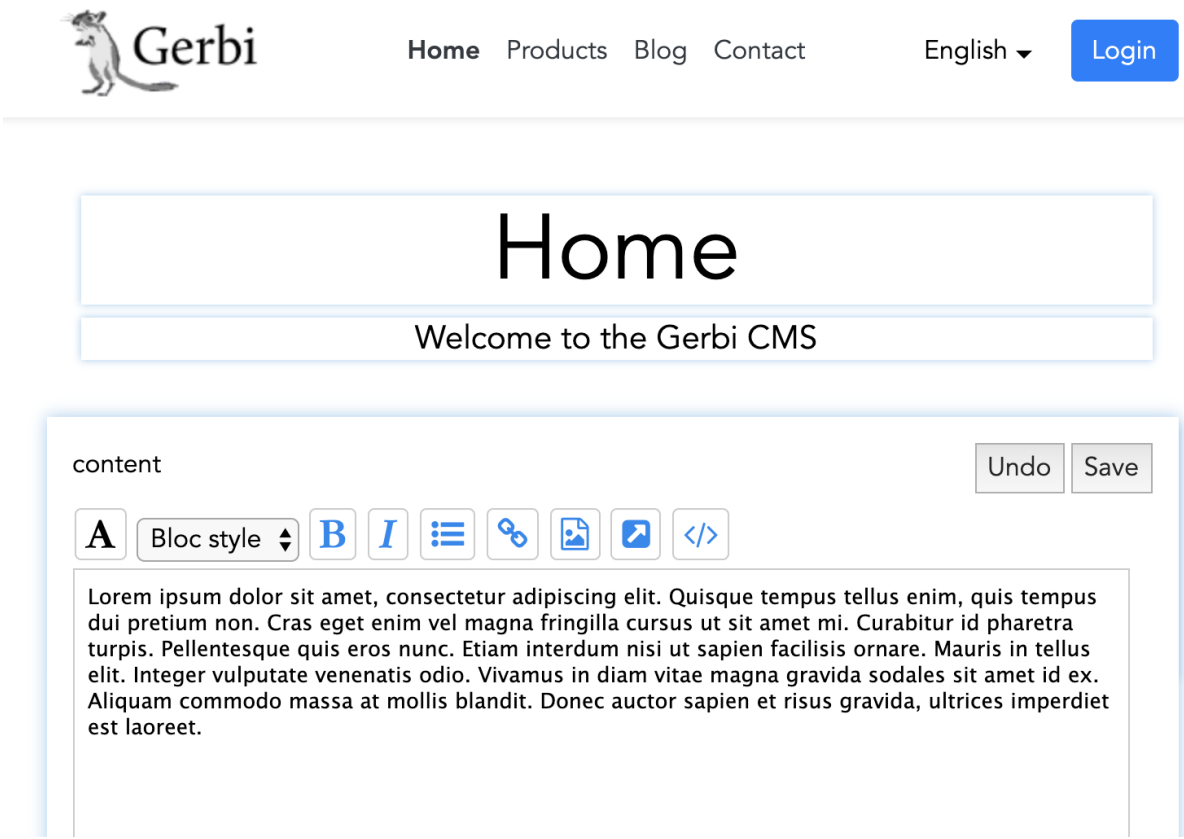
For your view to be used in place of the CMS one, you simply need to point to it with something similar to this:

```
from django.conf.urls import url
from YOUR_APP.views import details
from pages import page_settings

urlpatterns = []

if page_settings.PAGE_USE_LANGUAGE_PREFIX:
    urlpatterns += [
        url(r'^(?P<lang>[-\w]+)/(?P<path>.*)$', details,
            name='pages-details-by-path')
    ]
else:
    urlpatterns += [
        url(r'^(?P<path>.*)$', details, name='pages-details-by-path')
    ]
```

Note: Have a look at *pages.urls* for a up to date example of URLs configuration.

1.6 Inline page editing

Gerbi CMS provide a way to easily edit the placeholder's content on your own frontend when you are authenticated as a staff user.



To activate the feature on your frontend you need to add 2 tags inside your templates. One in your <head> tag and one in the <body> tag:

```
{% load pages_tags static i18n %}
<html>
    <head>
        ...
        {% pages_edit_media %}
    </head>
    <body>
        ...
        {% pages_edit_init %}
    </body>
</html>
```

For placeholder editing to work properly you will need to surround the placeholder with an HTML element like so:

```
<div>
    {% placeholder "title" %}
</div>
```

Placeholder editing works automatically by injecting an HTML comment into the placeholder output. So thing like this will not work:

```
<div>
    <img src="{{ MEDIA_URL }}{% imageplaceholder "img" %}">
</div>
```

Because the rendered comment will not result into an HTML node:

```
<div>
    <img src="upload/image.png<!--placeholder:img-->">
</div>
```

To fix this issue you can use placeholders as rendering blocks like so:

```
<div>
    {% imageplaceholder 'img' block %}
        {% if content %}
            <img src="{{ MEDIA_URL }}{{ content }}" class="img-responsive" alt="">
        {% endif %}
    {% endplaceholder %}
</div>
```

1.7 How to use the various navigation template tags

Presenting a navigational structure to the user is an common task on every website. Django-pages-cms offers various template tags which can be used to create a site navigation menu.

- *pages_navigation variable*
- *pages_menu*
- *pages_dynamic_tree_menu*
- *pages_sub_menu*
- *pages_siblings_menu*
- *pages_breadcrumb*
- *load_pages*

1.7.1 pages_navigation variable

By default the variable *pages_navigation* will be available within all of the CMS pages. *pages_navigation* is a list of Pages obtained by calling the *get_navigation* method on the *Details* class based view of the CMS:

```
def get_navigation(self, request, path, lang):
  """Get the pages that are at the root level."""
  return Page.objects.navigation().order_by("tree_id")
```

You can subclass the *Details* class based view to change this behaviour.

1.7.2 pages_menu

The pages_menu tag displays the whole navigation tree, including all subpages. This is useful for smaller sites which do not have a large number of pages.

Use the following snippet in your template:

```
<ul>
{% for page in pages_navigation %}
    {% pages_menu page %}
{% endfor %}
</ul>
```

The pages_menu tag uses the *pages/menu.html* template to render the navigation menu. By default, the menu is rendered as a nested list:

```
<ul>
    <li><a href="/page/1">page1</a></li>
    ...
</ul>
```

You can of course change *pages/menu.html* with the Django override mechanism to render things differently.

1.7.3 pages_dynamic_tree_menu

The pages_dynamic_tree_menu tag works similar to the pages_menu tag. but instead of displaying the whole navigation structure, only the following pages are displayed:

- all "root" pages (pages which have no parent)
- all parents of the current page
- all direct children of the current page

This type of navigation is recommended if your site has a large number of pages and/or a deep hierarchy, which is too complex or large to be presented to the user at once.

Use the following snippet in your template:

```
<ul>
{% for page in pages_navigation %}
    {% pages_dynamic_tree_menu page %}
{% endfor %}
</ul>
```

The pages_dynamic_tree_menu tag uses the *pages/dynamic_tree_menu.html* template to render the navigation menu. By default, the menu is rendered as a nested list similar to the pages_menu tag.

1.7.4 pages_sub_menu

The pages_sub_menu tag shows all the children of the root of the current page (the highest in the hierarchy). This is typically used for a secondary navigation menu.

Use the following snippet to display a list of all the children of the current root:

```
<ul>
{% pages_sub_menu current_page %}
</ul>
```

Again, the default template *pages/sub_menu.html* will render the items as a nested, unordered list (see above).

1.7.5 pages_siblings_menu

The pages_siblings_menu tag shows all the children of the immediate parent of the current page. This can be used for example as a secondary menu.

Use the following snippet to display a list of all the children of the immediate parent of the current page:

```
<ul>
{% pages_siblings_menu current_page %}
</ul>
```

Again, the default template *pages/sub_menu.html* will render the items as a nested, unordered list (see above).

1.7.6 pages_breadcrumb

With the pages_breadcrumb tag, it is possible to use the "breadcrumb"/"you are here" navigational pattern, consisting of a list of all parents of the current page:

```
{% pages_breadcrumb current_page %}
```

The output of the pages_breadcrumb tag is defined by the template *pages/breadcrumb.html*.

1.7.7 load_pages

The load_pages Tag can be used to load the navigational structure in views which are *not* rendered through page's own details() view. It will check the current template context and adds the pages and current_page variable to the context, if they are not present.

This is useful if you are using a common base template for your whole site, and want the pages menu to be always present, even if the actual content is not a page.

The load_pages does not take any parameters and must be placed before one of the menu-rendering tags:

```
{% load_pages %}
```

1.8 How to create a Blog application

A Blog is made of dated Blog Posts. It so happens this CMS pages model contain a creation date and is flexible enough to contain all sort contents that could be useful in a Blog post. So how can we use this CMS to create a Blog application? This guide gives a step by step recipe that demonstrate how easily you can build new features on top of this CMS.

1.8.1 Step 1: Create a new Django app within your project

```
python manage.py startapp blog
```

1.8.2 Step 2: Create the views

Open you newly created *blog/views.py* file and create 2 views. One for the category view and another for the Blog's index. We are using the *django-taggit* application to handle the categories, this way a Blog Post can be included in several categories.

```
from django.shortcuts import render
from pages.models import Page
from taggit.models import Tag
from django.core.paginator import Paginator

def category_view(request, *args, **kwargs):
    context = dict(kwargs)
    category = Tag.objects.get(id=kwargs['tag_id'])
    page = context['current_page']
    blogs = page.get_children_for_frontend().filter(tags__name__in=[category.name])

    paginator = Paginator(blogs, 8)
    page_index = request.GET.get('page')
    blogs = paginator.get_page(page_index)
    context['blogs'] = blogs

    context['category'] = category.name
    return render(request, 'blog-home.html', context)

def blog_index(request, *args, **kwargs):
    context = dict(kwargs)
    page = context['current_page']
    blogs = page.get_children_for_frontend()

    paginator = Paginator(blogs, 7)
    page = request.GET.get('page')
    blogs = paginator.get_page(page)
    context['blogs'] = blogs
    context['template_name'] = 'blog-home.html'

    return render(request, 'blog-home.html', context)
```

1.8.3 Step 2: Link the views with the CMS

Populate *blog/urls.py* with those urls

```
from django.conf.urls import url
from blog import views
from django.urls import include, path, re_path

urlpatterns = [
  url(r'^category/(?P<tag_id>[0-9]+)$', views.category_view, name='blog_category_view'),
  url(r'^$', views.blog_index, name='blog_index')
]
```

Then the last step is to register this URL module with the CMS. Place this code at the top of you project *urls.py* file.

```
from pages.urlconf_registry import register_urlconf
register_urlconf('blog', 'blog.urls', label='Blog index')
```

1.8.4 Step 3: Create the blog templates

You will need create 3 templates for your blog application. The first one is a helper template called *blog-card.html*. It contains a basic representation of a blog as a card:

```
{% load pages_tags static i18n humanize thumbnail %}
<div class="card mb-4 shadow-sm">
  <a class="blog-lead-image" href="{% show_absolute_url page %}">
    {% get_content page "lead-image" as image %}
    {% if image %}
      {% thumbnail image "320x240" crop="center" as img %}
        <img src="{{ img.url }}">
      {% endthumbnail %}
    {% else %}
      <!-- no image for this post -->
    {% endif %}
  </a>
  <div class="card-body">
    <a href="{% show_absolute_url page %}">
      <h3 class="my-0 font-weight-normal">{% show_content page "title" %}</h3>
    </a>
    <p>{% show_content page "lead" %}</p>
    {% if forloop.first %}
      {% get_content page "content" as content %}
      <p class="d-none d-lg-block">{{ content | striptags | safe | truncatechars:220 }}</
↪p>
    {% endif %}
    <p class="blog-meta">Published {{ page.creation_date | naturalday }}
      {% if page.tags.count %}
        in the categories:
        {% for tag in page.tags.all %}
          <a href="/{{ lang }}/blog/category/{{ tag.id }}">{{ tag.name }}</a>{% if not␣
↪forloop.last %},{% endif %}
        {% endfor %}
```

(continues on next page)

(continued from previous page)

```
      {% endif %}
      by {{ page.author.first_name }} {{ page.author.last_name }}
    </p>
  </div>
</div>
```

The second is the *blog-home.html* referenced by the views you previoulsy wrote, it will be used by the index and the categories:

```
{% extends 'index.html' %}
{% load pages_tags static i18n humanize thumbnail %}

{% block header %}
<div class="px-3 py-3 pt-md-5 pb-md-4 mx-auto text-center">
  <h1 class="display-4">{% placeholder "title" %} {{ category }}</h1>
  <p class="lead">{% placeholder "lead" with Textarea %}</p>
</div>
{% endblock %}

{% block content %}
  <div class="card-deck mb-3 text-center blog-home">
  {% for page in blogs %}
    {% include "blog-card.html" %}
  {% endfor %}
  </div>

  <div class="pagination">
      <span class="step-links">
          {% if blogs.has_previous %}
              <a href="?page=1" class="btn btn-light">&laquo; first</a>
              <a href="?page={{ blogs.previous_page_number }}" class="btn btn-light">
↪previous</a>
          {% endif %}

          <span class="current">
              Page {{ blogs.number }} of {{ blogs.paginator.num_pages }}.
          </span>

          {% if blogs.has_next %}
              <a href="?page={{ blogs.next_page_number }}" class="btn btn-light">next</a>
              <a href="?page={{ blogs.paginator.num_pages }}" class="btn btn-light">last␣
↪&raquo;</a>
          {% endif %}
      </span>
  </div>
  {% endblock %}
```

Finaly the last one is for the Blog Post itself. You could have different Blog Post templates but for now we only need one, let's call it *blog-post.html*:

```
{% extends 'index.html' %}
{% load pages_tags static i18n humanize %}
```

(continues on next page)

(continued from previous page)

```
{% block header %}
<div class="px-3 py-3 pt-md-5 pb-md-4 mx-auto text-center blog-post">
  <h1 class="display-4">{% placeholder "title" %}</h1>
  <p class="lead">{% placeholder "lead" with Textarea %}</p>
  <p class="blog-meta">Published {{ current_page.creation_date | naturalday }}
      {% if current_page.tags.count %}
        in the categories:
        {% for tag in current_page.tags.all %}
          <a href="/{{ lang }}/blog/category/{{ tag.id }}">{{ tag.name }}</a>{% if not␣
↪forloop.last %},{% endif %}
        {% endfor %}
      {% endif %}
      by {{ current_page.author.first_name }} {{ current_page.author.last_name }}
    </p>
</div>
{% endblock %}

{% block content %}
<div class="blog-post">
  <div class="blog-lead-image">
    {% imageplaceholder 'lead-image' block %}
      {% if content %}
        <img src="{{ MEDIA_URL }}{{ content }}" alt="">
      {% endif %}
    {% endplaceholder %}
  </div>

  <div>
    {% placeholder "content" with RichTextarea %}
  </div>
</div>
{% endblock %}
```

To finish things up you need to allow those 2 templates to be selected by the CMS. Add them to your *PAGE_TEMPLATES* setting:

```
PAGE_TEMPLATES = (
    ('index.html', 'Default template'),
    ('blog-post.html', 'Blog post'),
    ('blog-home.html', 'Blog home'),
)
```

1.8.5 Step 4: Activate the Blog in the admin

You can now activate the Blog in the CMS admin. To do so follow those few steps:

1. Create a new page named "Blog", chose the template "Blog Home", and the option "Delegate to application: Blog Index".
2. Add a couple of child pages to this Blog Index page and chose the "Blog Post" template as their template.

The result should be a functional blog with an index page, category pages, tagging and pagination.

You are also free to create serveral Blog instances withing the CMS by repeating a version of this step. There is no restrictions.

A fully functionnal version of this Blog application is available

1.9 Integrate with other applications

1.9.1 Delegate the rendering of a page to an application

By delegating the rendering of a page to another application, you will be able to use customized views and still get all the CMS variables to render a proper navigation.

First you need a *urls.py* file that you can register to the CMS. This file might look like this:

```
from django.conf.urls import url
from pages.testproj.documents.views import document_view

urlpatterns = [
    url(r'^doc-(?P<document_id>[0-9]+)$', document_view, name='document_details'),
    url(r'^$', document_view, name='document_root'),
]
```

Note: The CMS will pass the keyword arguments *current_page*, *pages_navigation* and *lang* to the view

Then you need to register the urlconf module with the CMS to use it within the admin interface. Here is an example for a document application.:

```
from pages.urlconf_registry import register_urlconf

register_urlconf('Documents', 'pages.testproj.documents.urls',
    label='Display documents')
```

As soon as you have registered your *urls.py*, a new field will appear in the page administration. Choose the *Display documents*. The view used to render this page on the frontend is now choosen by *pages.testproj.documents.urls*.

Note: The path passed to your urlconf module is the remaining path available after the page slug. Eg: */pages/page1/doc-1* will become *doc-1*.

Note: If you want to have the reverse URLs with your delegated application, you will need to include your URLs into your main urls.py, eg:

```
(r'^pages/', include('pages.urls')),
...
(r'^pages/(?P<path>.*)', include('pages.testproj.documents.urls')),
```

Here is an example of a valid view from the documents application:

```
from django.shortcuts import render
from pages.testproj.documents.models import Document

def document_view(request, *args, **kwargs):
    context = dict(kwargs)
    if kwargs.get('current_page', False):
        documents = Document.objects.filter(page=kwargs['current_page'])
        context['documents'] = documents
    if kwargs.has_key('document_id'):
        document = Document.objects.get(pk=int(kwargs['document_id']))
        context['document'] = document
    context['in_document_view'] = True
    return render(request, 'pages/examples/index.html', context)
```

The *document_view* will receive a bunch of extra parameters related to the CMS:

- *current_page* the page object,
- *path* the path used to reach the page,
- *lang* the current language,
- *pages_navigation* the list of pages used to render navigation.

Note: If the field doesn't appear within the admin interface make sure that your registry code is executed properly.

Note: Look at the testproj in the repository for an example on how to integrate an external application.

1.9.2 Sitemaps

Gerbi CMS provide 2 sitemaps classes to use with Django sitemap framework:

```
from pages.views import PageSitemap, MultiLanguagePageSitemap

(r'^sitemap\.xml$', 'django.contrib.sitemaps.views.sitemap',
    {'sitemaps': {'pages':PageSitemap}}),

# or for multi language:

(r'^sitemap\.xml$', 'django.contrib.sitemaps.views.sitemap',
    {'sitemaps': {'pages':MultiLanguagePageSitemap}})
```

The *PageSitemap* class provide a sitemap for every published page in the default language. The *MultiLanguagePageSitemap* is gonna create an extra entry for every other language.

1.10 Commands

Contents

1.10.1 Import and export content between diffent hosts using the API

Gerbi CMS comes with a simple API that uses Django REST Framework. You can enable the API by setting the PAGE_API_ENABLED in the CMS settings to *True*.

When this is done, your host should have an API at the following address <address of your cms>/api/

Pulling data from a host: pages_pull

From example if you enabled the API on your local development instance you can do:

```
$ python manage.py pages_pull staff_account_name:password
Fetching page data on http://127.0.0.1:8000/api/
data/download.json written to disk
```

The default for the host is the localhost (127.0.0.1:8000). This command accept the option *–filename* and *–host*

Pushing data to a host: pages_push

Similarly you can push the collected data to an host:

```
python manage.py pages_push staff_account_name:password
Fetching the state of the pages on the server http://127.0.0.1:8000/api/
Update page 1 ............
Update page 2 ....
Update page 3 ....
```

This command accepts the option *–filename* and *–host*

Limitations

The push command does it's best creating and updating pages between 2 hosts but equivalency cannot be guaranteed. Also files associated with the pages are yet not transfered using this API.

To best syncronise 2 hosts (A:production, B: staging) first pull from A and push the content to B to have an accurate representation of production data.

Do your modification on B then then push back on A.

1.10.2 Export pages content into translatable PO files

The pages CMS provide a command for those that would prefer to use PO files instead of the admin inteface to translate the pages content.

To export all the content from the published page into PO files you can execute this Django command:

```
$ python manage.py pages_export_po <path>
```

The files a created in the *poexport* directory if no path is provided.

After the translation is done, you can import back the changes with another command:

```
$ python manage.py pages_import_po <path>
```

1.11 List of all available settings

1.11.1 PAGE_TEMPLATES

PAGE_TEMPLATES is a list of tuples that specifies the which templates are available in the `pages` admin. Templates should be assigned in the following format:

```
PAGE_TEMPLATES = (
    ('pages/nice.html', 'nice one'),
    ('pages/cool.html', 'cool one'),
)
```

One can also assign a callable (which should return the tuple) to this setting to achieve dynamic template list e.g.:

```
def _get_templates():
    # to avoid any import issues
    from app.models import PageTemplate
    return PageTemplate.get_page_templates()

PAGE_TEMPLATES = _get_templates
```

Where the model might look like this:

```
class PageTemplate(OrderedModel):
    name = models.CharField(unique=True, max_length=100)
    template = models.CharField(unique=True, max_length=260)

    @staticmethod
```

(continues on next page)

(continued from previous page)

```
    def get_page_templates():
        return PageTemplate.objects.values_list('template', 'name')

    class Meta:
        ordering = ["order"]

    def __unicode__(self):
        return self.name
```

Note: You can use django-dbtemplates to store the actual template in your database if it what you want.

Warning: pages checks that every template in PAGE_TEMPLATE exists with `django.template.loader.find_template` when its app config is ready. It will raise a warning `basic_cms.W001` if a template can't be found and if a template has a syntax error then a TemplateSyntaxError is raised.

1.11.2 PAGE_DEFAULT_TEMPLATE

You *must* set `PAGE_DEFAULT_TEMPLATE` to the path of your default template:

```
PAGE_DEFAULT_TEMPLATE = 'pages/index.html'
```

1.11.3 PAGE_LANGUAGES

A list tuples that defines the languages that pages can be translated into:

```
gettext_noop = lambda s: s

PAGE_LANGUAGES = (
    ('zh-cn', gettext_noop('Chinese Simplified')),
    ('fr', gettext_noop('Français')),
    ('en', gettext_noop('US English')),
)
```

1.11.4 PAGE_DEFAULT_LANGUAGE

Defines which language should be used by default. If `PAGE_DEFAULT_LANGUAGE` not specified, then project's `settings.LANGUAGE_CODE` is used:

```
LANGUAGE_CODE = 'en'
```

1.11.5 PAGE_LANGUAGE_MAPPING

PAGE_LANGUAGE_MAPPING must be a function that takes the language code of the incoming browser as an argument.

This function can change the incoming client language code to another language code, presumably one for which you are managing trough the CMS.

This is useful if your project only has one set of translation strings for a language like Chinese, which has several variants like `zh-cn`, `zh-tw`, `zh-hk`, but you don't have a translation for every variant.

`PAGE_LANGUAGE_MAPPING` help you to server the same Chinese translation to all those Chinese variants, not just those with the exact `zh-cn` locale.

Enable that behavior here by assigning the following function to the `PAGE_LANGUAGE_MAPPING` variable:

```
# here is all the languages supported by the CMS
PAGE_LANGUAGES = (
    ('de', gettext_noop('German')),
    ('fr-fr', gettext_noop('Français')),
    ('en', gettext_noop('US English')),
)

# copy PAGE_LANGUAGES
languages = list(PAGE_LANGUAGES)

# Other languages accepted as a valid client language
languages.append(('fr-fr', gettext_noop('French')))
languages.append(('fr-be', gettext_noop('Belgium french')))

# redefine the LANGUAGES setting in order to be sure to have the correct request.
↪LANGUAGE_CODE
LANGUAGES = languages

# Map every french based language to fr-fr
def language_mapping(lang):
    if lang.startswith('fr'):
        return 'fr-fr'
    return lang
PAGE_LANGUAGE_MAPPING = language_mapping
```

1.11.6 PAGES_MEDIA_URL

URL that handles pages media. If not set the default value is:

```
<STATIC_URL|MEDIA_URL>pages/
```

1.11.7 PAGE_UNIQUE_SLUG_REQUIRED

Set `PAGE_UNIQUE_SLUG_REQUIRED` to `True` to enforce unique slug names for all pages.

1.11.8 PAGE_CONTENT_REVISION

Set `PAGE_CONTENT_REVISION` to `False` to disable the recording of pages revision information in the database

1.11.9 SITE_ID

Set SITE_ID to the id of the default `Site` instance to be used on installations where content from a single installation is served on multiple domains via the `django.contrib.sites` framework.

1.11.10 PAGE_USE_SITE_ID

Set PAGE_USE_SITE_ID to `True` to make use of the `django.contrib.sites` framework.

1.11.11 PAGE_USE_LANGUAGE_PREFIX

Set PAGE_USE_LANGUAGE_PREFIX to `True` to make the `get_absolute_url` method to prefix the URLs with the language code (Default: False)

1.11.12 PAGE_CONTENT_REVISION_EXCLUDE_LIST

Assign a list of placeholders to PAGE_CONTENT_REVISION_EXCLUDE_LIST to exclude them from the revision process. (Default: [])

1.11.13 PAGE_HIDE_ROOT_SLUG

Hide the slug's of the first root page ie: `/home/` becomes `/`. (Default: False)

1.11.14 PAGE_SHOW_START_DATE

Show the publication start date field in the admin. Allows for future dating Changing the `PAGE_SHOW_START_DATE` from `True` to `False` after adding data could cause some weirdness. If you must do this, you should update your database to correct any future dated pages. (Default: False)

1.11.15 PAGE_SHOW_END_DATE

Show the publication end date field in the admin, allows for page expiration Changing `PAGE_SHOW_END_DATE` from `True` to `False` after adding data could cause some weirdness. If you must do this, you should update your database and null any pages with `publication_end_date` set. (Default: False)

1.11.16 PAGE_TAGGING

Set `PAGE_TAGGING` to `True` if you wish to use the `django-taggit` application. (Default: False)

1.11.17 PAGE_TESTS_SAVE_SCREENSHOTS

Allows you to save screenshots from selenium tests (Default: False)

1.11.18 PAGE_REDIRECT_OLD_SLUG

Allows to redirect to new url after updating slug (Default: False)

1.12 Changelog

This file describe new features and incompatibilities between released version of the CMS.

1.12.1 Release 2.0.12

- Support Django 3

1.12.2 Release 2.0.10

- manage.py command needs python 3

1.12.3 Release 2.0.9

- Gerbi command line need python 3

1.12.4 Release 2.0.8

- New translations, and docker compose examples

1.12.5 Release 2.0.7

- Bug fixes in admin drag and drop and cookie handling

1.12.6 Release 2.0.6

Released the 17th of January 2019.

- Improved inline editor CSS
- Update the example application to contain a blog example application using the 3rd party delegation feature.
- Update gerbi command line to contain the blog

1.12.7 Release 2.0.3

Released the 8th of January 2019.

- Fix 3rd party delegation (was broken after Django 2.0)

1.12.8 Release 2.0.2

Released the 7th of January 2019.

- Update admin icons to use SVG

1.12.9 Release 2.0.1

Released the 20th of December 2018.

- Fix allow tags in media section so there is an image preview

1.12.10 Release 2.0.0

Released the second of November 2018.

- Migrate to Django 2.1.X
- Removal of get_pages_with_tag

1.12.11 Release 1.9.17

Released the 9th of March 2017.

- There was an issue with Django-compressor and placeholders that needed fixing https://github.com/django-compressor/django-compressor/pull/825

1.12.12 Release 1.9.15

Released the 7th of February 2017.

- Bugfix for Django 1.11

1.12.13 Release 1.9.14

Released the 23rd of January 2017.

- Bugfix the demo for Django 1.10

1.12.14 Release 1.9.13

Released the 23rd of January 2017.

- Added a Media model for the new media library
- Added a way to insert images from the media library into the basic Rich Text Editor
- File placeholders now create a new media for each user upload

1.12.15 Release 1.9.12

Released the 5th of January 2017.

- Bugfix

1.12.16 Release 1.9.11

Released the 5th of January 2017.

- Improve the frontend edit mode with an edit bar on the top left of the page so you can disable the edit mode if necessary.

1.12.17 Release 1.9.10

Released the 27th of December 2016.

- Fix a performance issue with new placeholder block parsing. *Placeholder can now be used as blocks*

1.12.18 Release 1.9.9

Released the 18th of December 2016.

- Fix a couple of problems with inline frontend editing
- *Placeholder can now be used as blocks*

1.12.19 Release 1.9.8

Released the 13th of December 2016.

- Official support for inline frontend editing
- New shared keyword for placeholder: shared content accross pages

1.12.20 Release 1.9.7

Released the 23rd of October 2016.

- Fix the build and some details in the admin
- Improvement in the drag and drop interface

1.12.21 Release 1.9.6

Released the 11th of September 2016.

- Improvement in the page edit form UX

1.12.22 Release 1.9.5

Released the 8th of September 2016.

- Improvement in the drag and drop UX

1.12.23 Release 1.9.4

Released the 2nd of September 2016.

- Changes in setup.py so dependecies can be installed with *pip install django-page-cms[full]*

1.12.24 Release 1.9.3

Released the 2nd of Semptember 2016.

- A new conditional template tag called page_has_content
- A new gerbi console command to create demo websites: gerbi –create mywebsite
- Improve page admin look and feel
- Fix problems withing the admin (Javascript errors)
- Update documentation

1.12.25 Release 1.9.1

Released the 12th of June 2016.

- A new keyword section on the placeholer has been added to create groups in the admin
- Language fallback for empty page placeholders in the admin was enabeled and causing possible weirdness
- Grappelli support (incomplete)
- Support for section (grouping) fields in admin
- Clean upload file names
- Bug fixes
- Basic RTE improvement in the admin
- Code cleanup

1.12.26 Release 1.9.0

Released the 1st of February 2016.

- Support Django 1.9
- Support Python 3.4, 3.5
- Redirect to new urls after updating slug. New settings PAGE_REDIRECT_OLD_SLUG
- Added get_pages_with_tag templatetag
- Added tags in JSON export/import
- Code cleanup
- PAGE_CACHE_LOCATION setting is removed

1.12.27 Release 1.8.2

Released the 20th of December 2015.

- Migrations files were not included in 1.8.1
- Add a pages_push and pages_pull command that permit to pull and push content between different hosts in rather smart and non breaking way.

1.12.28 Release 1.8.1

Released the 24th of September 2015.

- Added support for a REST API using Django Rest Framework (http://www.django-rest-framework.org/)
- Refactoring

1.12.29 Release 1.8.0

Released the 23rd of April 2015.

- Updated to Django 1.8
- Jumped 2 version to stick with Django versionning numbers

Backward Incompatible Changes

- Incompatible with Django 1.7 and lower

1.12.30 Release 1.6.2

Released the 27th of July 2014.

- Added a ckeditor placeholder using django-ckeditor
- The project now use transifex to handle it's translations (https://www.transifex.com/projects/p/django-page-cms-1/)
- Fix several bugs related to placeholders and cache.
- Fix a bug with files using non ascii characters.
- Fix a bug with the loading icon.

1.12.31 Release 1.6.1

Released the 2nd of June 2014.

- Fix a bug with the image upload.
- Fix a bug with files using non ascii characters.
- Fix a bug with the loading icon.

1.12.32 Release 1.6.0

Released the 11th of March 2014.

Highlights

- Full compatibility with Python 3.3 (https://travis-ci.org/batiste/django-page-cms) as well python 2.7 with the same code base.
- Django-page-cms is now compatible with Django 1.6.2
- Setup selenium tests infrastructure
- *New Markdown Placeholder*
- Django-page-cms has a test coverage of 90%. Commits that bring this number down will be rejected.
- Preserve the language choice across saves in the admin interface
- Move the JSON export in it's own plugin application

Backward Incompatible Changes

- HTML sanitization and the dependecy to the html5lib have been removed.
- Remove support for WYMEditor, markItUp and CKEditor editors. Rational: Those Widget are untested, not updated and were created when packages for those widgets didn't exists as python packages (django-ckeditor, django-wymeditor, django-markitup). If you need those editors please install the package and register the widget to use them directly in your templates.
- The pages_navigation context processor has been removed. This is not useful as {% load_pages %} already load the pages_navigation variable in the context.
- Removal of the video placeholder. Rational: Used as an example but add no real value to the CMS.
- Removal of PageAdminWithDefaultContent. Rational: PageAdminWithDefaultContent is completly untested and can be easily reproduced in any project if necessary.
- Move po import/export to it's own plugin application.
- PAGE_CONNECTED_MODELS is gone. Use inline admin objects instead https://docs.djangoproject.com/en/dev/ref/contrib/admin/#inlinemodeladmin-objects

1.12.33 Release 1.5.3

Released the 23 of October 2013.

- Tiny MCE javascript is not included with this CMS anymore. Please use https://github.com/aljosa/django-tinymce
- A more agressive cache should reduce page related SQL queries to 3 once the cache is warm.
- A plugin app example as been created in pages.plugins.category.
- jquery.query-2.1.7.js is properly restaured this time.

1.12.34 Release 1.5.2

Released on the first of September 2013.

- Fix bad migrations.
- Test and fix a bug with the PAGE_AUTOMATIC_SLUG_RENAMING option.
- Re-introduce a previously delete javascript file (jquery.query-2.1.7.js) necessary in the admin interface.
- File and Image placeholer now use the same filename scheme that preserve the original filename.

1.12.35 Release 1.5.1

Released on the 7th of August 2013.

- Documentation fixes.
- Dependencies on html5lib was incorrect.
- Placeholder names can now be any string if quotes are used. "éà àü" is a valid placeholder name.

1.12.36 Release 1.5.0

- Full compatibility with Django 1.5
- New Drag and Drop interaction in the admin (jquery.ui not needed anymore)
- New placeholder JsonPlaceholderNode
- New settings PAGE_IMPORT_ENABLED, PAGE_EXPORT_ENABLED and PAGE_AUTOMATIC_SLUG_RENAMING
- Haystack 2.0 compatibility (not tested)
- Cleanup the admin JavaScript files
- Possibility to Substituting a custom User model (new in Django 1.5)
- Remove the dependency on BeautifulSoup

1.12.37 Release 1.4.3

- New placeholder tag: contactplaceholder that produce a contact form.
- Performance improvement: don't render the template with a Context in the get_placeholder method.
- Fix some issue with Ajax calls and csrf protection.
- Fix some outdated migrations.
- New placeholder tag: fileplaceholder allows users to upload files.
- Italian traduction.
- Added X-View headers to response in order to work with 'Edit this object' bookmarklet.

1.12.38 Release 1.4.2

- Fix a packaging issue with the static files. The package_data setup variable was incorrect.

1.12.39 Release 1.4.1

- Tests are not executed when you execute ./manage.py test, unless explicity enabled with PAGE_ENABLE_TESTS.
- Deprecation of the auto_render decorator.
- Fix the request mock to work with the latest trunk of Django.
- ImagePlaceholder: use django.core.files.storage.default_storage instead of from django.core.files.storage import FileSystemStorage
- Added setting for allowing realtime search index rather than index on management command.
- Optimize and cache is_first_root method.
- Fix a bug in the {% get_content %} tag.

1.12.40 Release 1.4.0

- A cute new name for the django page CMS : *Gerbi CMS*. The package name will remain *django-page-cms* for this release but might be changed to *gerbi* in a near future.
- Implement 2 classes for the Django sitemap framework. *Documentation on sitemap classes*
- Add a markitup REST editor.
- Fix a bug with *pages_dynamic_tree_menu* template tag and multiple roots in a pages tree.
- Added a PAGES_STRICT_URLS setting. If set to *True* the CMS will check for the complete URL instead of just the slug. If the complete path doesn't match, a 404 error is raised in the view.
- Added 2 managing commands for exporting and importing PO translation files into the CMS. *Documentation on the commands*
- Add a PAGE_CONTENT_REVISION_DEPTH setting to limit the amount of revision we want to keep.
- Fix a bug so the CMS can run without django-taggit installed.
- Fix a bug with placeholder and template inheritance.
- The *pages-root* URL doesn't need to be specified anymore. But you can still use it if you want to define a special URL for the root page.

Backward Incompatible Changes

- New delegation rules: the CMS delegate not only the exact path leading to the page but also the whole sub path. *Documentation on the delegation as been updated*.
- The default view now raise an *ValueError* if the *path* argument is not passed instead of guessing the path by using *request.path*.

1.12.41 Release 1.3.0

- The default view is now a class therefor you can subclass it and change it's behavior more easily.
- Fix a bug with get_slug_relative_path that may strip the language 2 times from the URL.
- Remove the dependency to django-unittest-depth.
- Don't raise a 404 when the LANGUAGE_CODE language is not present in the PAGE_LANGUAGES list.
- Get ride of the only raw SQL command by using the ORM's annotate.
- Fix a cache issue with show_absolute_url and get_complete_slug.
- The default template for menu now display the title instead of the slug in the link
- Improve the default application look.

Incompatible changes

- Placeholer content is now marked as safe by default.
- The CMS need the new version of django-mptt 0.4.1.
- Remove the support for django-tagging and use django-taggit instead.

Maintenance

Install the new django-mptt package:

```
sudo pip install -U django-mptt>=0.4.1
```

If you want to use tags you should install the new django-taggit:

```
sudo pip install django-taggit
```

1.12.42 Release 1.2.1

- Change the cache class attributes into data attributes as it was intented in the design for the "per instance" cache.

1.12.43 Release 1.2.0

- Add publish right managements in the admin.
- Fix an admin bug with the untranslated option for placeholder.
- Fix the package so the media are included.
- Fix bug with the default value of PAGE_TEMPLATES doesn't trigger an error in the admin when unspecified.
- Add a delete image feature to the image placeholder.
- Make root page url '/' work with the PAGE_USE_LANGUAGE_PREFIX option.
- Change the placeholder save prototype by adding an extra keyword parameter: extra_data.
- Fix a bug with the image placeholder when the "save and continue" button is used.

1.12.44 Release 1.1.3

- Improved search index (url and title are included).
- The setup now specify django-mptt-2 instead of django-mptt.
- New template tag for navigation called "pages_siblings_menu".
- New object PageAdminWithDefaultContent: copy the official language text into new language page's content blocks
- New setting PAGE_HIDE_SITES to hide the sites. When True the CMS only show pages from the current site used to access the admin. This allows administration of separate page-cms sites with the same DB.
- New admin template tag: language_content_up_to_date templatetag: mark the translations needing updating in the admin.

- DEFAULT_PAGE_TEMPLATE is rennomed into PAGE_DEFAULT_TEMPLATE. This setting will still continue to work.
- Add a new template tag get_page to insert page object into the context.

1.12.45 Release 1.1.2

- Change the default value of PAGE_TAGGING and PAGE_TINYMCE to *False*
- Implement drag and drop within the admin interface.
- Implement haystack SearchIndex for page content search.
- Add the untranslated placeholder keyword. Enable the user to have a single placeholder content accross all languages.
- Add back the hierarchical change rights management for every page.

1.12.46 Release 1.1.1

- Add new inherited placeholder option to inherit content from a parent page.
- PagePermission object is gone in favor of django-authority.
- New permission by language.
- New permission for freezing page content.
- Add a get_date_ordered_children_for_frontend Page's method.
- Add missing templates to the package.

1.12.47 Release 1.1.0

- PAGE_TEMPLATES setting can also be a callable.
- PAGE_UPLOAD_ROOT setting enable you to choose where files are uploaded.
- The CMS comes with south migrations if you want to use them.
- *get_url* is renamed into *get_complete_slug*.
- *get_absolute_url* is renamed into *get_url_path*.
- Admin widgets now needs to use a registery to be used within the admin. The placeholder template tag doesn't load load external modules for you anymore.
- RTL support for pages in admin.
- The context variable *pages* has been renamed to *pages_naviagtion* to avoid any name conflict with some pagination tags.

Maintenance

A new character field called *delegate_to* is added to the page model. to enable the delegation of the pages rendering to a 3rd party application:

```
ALTER TABLE pages_page ADD COLUMN delegate_to varchar(100) NULL;
```

1.12.48 Release 1.0.9

- Finish to migrate the old wiki into the sphinx documentation
- Fix the package so it can be installed properly with easy_install
- Add a new placeholder {% imageplaceholder %} for a basic automatic image handling in the admin.

1.12.49 Release 1.0.8

- A few bug fix.
- A automatic internal link system. Page link don't break even if you move the linked page.

1.13 Page CMS reference API

1.13.1 The application model

Gerbi CMS declare rather simple models: *Page* *Content* and *PageAlias*.

1.13.2 Placeholders

Placeholder module, that's where the smart things happen.

class pages.placeholders.**ContactForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *field_order=None*, *use_required_attribute=None*, *renderer=None*)

Simple contact form

base_fields = {'email': <django.forms.fields.EmailField object>, 'message': <django.forms.fields.CharField object>, 'subject': <django.forms.fields.CharField object>}

declared_fields = {'email': <django.forms.fields.EmailField object>, 'message': <django.forms.fields.CharField object>, 'subject': <django.forms.fields.CharField object>}

property media

Return all media required to render the widgets on this form.

class pages.placeholders.**ContactPlaceholderNode**(*name*, *page=None*, *widget=None*, *parsed=False*, *as_varname=None*, *inherited=False*, *untranslated=False*, *has_revision=True*, *section=None*, *shared=False*, *nodelist=None*)

A contact *PlaceholderNode* example.

render(*context*)

Output the content of the *PlaceholdeNode* as a template.

class pages.placeholders.**FilePlaceholderNode**(*name*, *page=None*, *widget=None*, *parsed=False*, *as_varname=None*, *inherited=False*, *untranslated=False*, *has_revision=True*, *section=None*, *shared=False*, *nodelist=None*)

A *PlaceholderNode* that saves one file on disk.

PAGE_UPLOAD_ROOT setting define where to save the file.

get_field(*page*, *language*, *initial=None*)

The field that will be shown within the admin.

save(*page*, *language*, *data*, *change*, *extra_data=None*)

Actually save the placeholder data into the Content object.

class pages.placeholders.**ImagePlaceholderNode**(*name*, *page=None*, *widget=None*, *parsed=False*, *as_varname=None*, *inherited=False*, *untranslated=False*, *has_revision=True*, *section=None*, *shared=False*, *nodelist=None*)

A *PlaceholderNode* that saves one image on disk.

PAGE_UPLOAD_ROOT setting define where to save the image.

get_field(*page*, *language*, *initial=None*)

The field that will be shown within the admin.

class pages.placeholders.**JsonPlaceholderNode**(*name*, *page=None*, *widget=None*, *parsed=False*, *as_varname=None*, *inherited=False*, *untranslated=False*, *has_revision=True*, *section=None*, *shared=False*, *nodelist=None*)

A *PlaceholderNode* that try to return a deserialized JSON object in the template.

get_render_content(*context*)

class pages.placeholders.**MarkdownPlaceholderNode**(*name*, *page=None*, *widget=None*, *parsed=False*, *as_varname=None*, *inherited=False*, *untranslated=False*, *has_revision=True*, *section=None*, *shared=False*, *nodelist=None*)

A *PlaceholderNode* that return HTML from MarkDown format

render(*context*)

Render markdown.

widget

alias of `Textarea`

class pages.placeholders.**PlaceholderNode**(*name*, *page=None*, *widget=None*, *parsed=False*, *as_varname=None*, *inherited=False*, *untranslated=False*, *has_revision=True*, *section=None*, *shared=False*, *nodelist=None*)

This template node is used to output and save page content and dynamically generate input fields in the admin.

Parameters

- **name** – the name of the placeholder you want to show/create
- **page** – the optional page object
- **widget** – the widget you want to use in the admin interface. Take a look into *pages.widgets* to see which widgets are available.
- **parsed** – if the `parsed` word is given, the content of the placeholder is evaluated as template code, within the current context.
- **as_varname** – if `as_varname` is defined, no value will be returned. A variable will be created in the context with the defined name.
- **inherited** – inherit content from parent's pages.
- **untranslated** – the placeholder's content is the same for every language.

edit_tag()

field

alias of `CharField`

get_content(*page_obj*, *lang*, *lang_fallback=True*)

get_content_from_context(*context*)

get_extra_data(*data*)

Get eventual extra data for this placeholder from the admin form. This method is called when the Page is saved in the admin and passed to the placeholder save method.

get_field(*page*, *language*, *initial=None*)

The field that will be shown within the admin.

get_lang(*context*)

get_render_content(*context*)

get_widget(*page*, *language*, *fallback=<class 'django.forms.widgets.Textarea'>*)

Given the name of a placeholder return a *Widget* subclass like Textarea or TextInput.

render(*context*)

Output the content of the *PlaceholdeNode* as a template.

render_parsed(*context*, *content*)

save(*page*, *language*, *data*, *change*, *extra_data=None*)

Actually save the placeholder data into the Content object.

widget

alias of `TextInput`

pages.placeholders.**get_filename**(*page*, *content_type*, *data*)

Generate a stable filename using the original filename of the type.

pages.placeholders.**parse_placeholder**(*parser*, *token*)

Parse the *PlaceholderNode* parameters.

Return a tuple with the name and parameters.

1.13.3 Template tags

Page CMS page_tags template tags

class pages.templatetags.pages_tags.**GetContentNode**(*page*, *content_type*, *varname*, *lang*, *lang_filter*)

Get content node

render(*context*)

Return the node rendered as a string.

class pages.templatetags.pages_tags.**GetPageNode**(*page_filter*, *varname*)

get_page Node

render(*context*)

Return the node rendered as a string.

class pages.templatetags.pages_tags.**LoadEditMediaNode**

Load edit node.

render(*context*)

Return the node rendered as a string.

class pages.templatetags.pages_tags.**LoadEditNode**

Load edit node.

render(*context*)

Return the node rendered as a string.

class pages.templatetags.pages_tags.**LoadPagesNode**

Load page node.

render(*context*)

Return the node rendered as a string.

pages.templatetags.pages_tags.**do_contactplaceholder**(*parser*, *token*)

Method that parse the contactplaceholder template tag.

pages.templatetags.pages_tags.**do_fileplaceholder**(*parser*, *token*)

Method that parse the fileplaceholder template tag.

pages.templatetags.pages_tags.**do_get_content**(*parser*, *token*)

Retrieve a Content object and insert it into the template's context.

Example:

```
{% get_content page_object "title" as content %}
```

You can also use the slug of a page:

```
{% get_content "my-page-slug" "title" as content %}
```

Syntax:

```
{% get_content page type [lang] as name %}
```

Parameters

- **page** – the page object, slug or id
- **type** – content_type used by a placeholder
- **name** – name of the context variable to store the content in
- **lang** – the wanted language

pages.templatetags.pages_tags.**do_get_page**(*parser*, *token*)

Retrieve a page and insert into the template's context.

Example:

```
{% get_page "news" as news_page %}
```

Parameters

- **page** – the page object, slug or id
- **name** – name of the context variable to store the page in

pages.templatetags.pages_tags.**do_imageplaceholder**(*parser*, *token*)

Method that parse the imageplaceholder template tag.

pages.templatetags.pages_tags.**do_jsonplaceholder**(*parser*, *token*)

Method that parse the contactplaceholder template tag.

pages.templatetags.pages_tags.**do_load_edit**(*parser*, *token*)

pages.templatetags.pages_tags.**do_load_edit_media**(*parser*, *token*)

pages.templatetags.pages_tags.**do_load_pages**(*parser*, *token*)

Load the navigation pages, lang, and current_page variables into the current context.

Example:

```
<ul>
    {% load_pages %}
    {% for page in pages_navigation %}
        {% pages_menu page %}
    {% endfor %}
</ul>
```

pages.templatetags.pages_tags.**do_markdownlaceholder**(*parser*, *token*)

Method that parse the markdownplaceholder template tag.

pages.templatetags.pages_tags.**do_page_has_content**(*parser*, *token*)

Conditional tag that only renders its nodes if the page has content for a particular content type. By default the current page is used.

Syntax:

```
{% page_has_content <content_type> [<page var name>] %}
    ...
{%_end page_has_content %}
```

Example use:

```
{% page_has_content 'header-image' %}
    <img src="{{ MEDIA_URL }}{% imageplaceholder 'header-image' %}">
{% end_page_has_content %}
```

pages.templatetags.pages_tags.**do_placeholder**(*parser*, *token*)

Method that parse the placeholder template tag.

Syntax:

```
{% placeholder <name> [on <page>] [with <widget>] [parsed] [as <varname>] %}
```

Example usage:

```
{% placeholder about %}
{% placeholder body with TextArea as body_text %}
{% placeholder welcome with TextArea parsed as welcome_text %}
{% placeholder teaser on next_page with TextArea parsed %}
```

pages.templatetags.pages_tags.**get_page_from_string_or_id**(*page_string*, *lang=None*)

Return a Page object from a slug or an id.

pages.templatetags.pages_tags.**has_content_in**(*page*, *language*)

Fitler that return `True` if the page has any content in a particular language.

Parameters

- **page** – the current page
- **language** – the language you want to look at

pages.templatetags.pages_tags.**language_content_up_to_date**(*page*, *language*)

Tell if all the page content has been updated since the last change of the official version (settings.LANGUAGE_CODE)

This is approximated by comparing the last modified date of any content in the page, not comparing each content block to its corresponding official language version. That allows users to easily make "do nothing" changes to any content block when no change is required for a language.

pages.templatetags.pages_tags.**pages_admin_menu**(*context*, *page*)

Render the admin table of pages.

pages.templatetags.pages_tags.**pages_breadcrumb**(*context*, *page*, *url='/'*)

Tags

pages.templatetags.pages_tags.**pages_dynamic_tree_menu**(*context*, *page*, *url='/'*)

Render a "dynamic" tree menu, with all nodes expanded which are either ancestors or the current page itself.

Override `pages/dynamic_tree_menu.html` if you want to change the design.

Parameters

- **page** – the current page
- **url** – not used anymore

pages.templatetags.pages_tags.**pages_menu**(*context*, *page*, *url='/'*)

Render a nested list of all the descendents of the given page, including this page.

Parameters

- **page** – the page where to start the menu from.
- **url** – not used anymore.

pages.templatetags.pages_tags.**pages_siblings_menu**(*context*, *page*, *url='/'*)

Get the parent page of the given page and render a nested list of its child pages. Good for rendering a secondary menu.

Parameters

- **page** – the page where to start the menu from.
- **url** – not used anymore.

pages.templatetags.pages_tags.**pages_sub_menu**(*context*, *page*, *url='/'*)

Get the root page of the given page and render a nested list of all root's children pages. Good for rendering a secondary menu.

Parameters

- **page** – the page where to start the menu from.
- **url** – not used anymore.

pages.templatetags.pages_tags.**show_absolute_url**(*context*, *page*, *lang=None*)

Show the url of a page in the right language

Example

```
{% show_absolute_url page_object %}
```

You can also use the slug of a page:

```
{% show_absolute_url "my-page-slug" %}
```

Keyword arguments: :param page: the page object, slug or id :param lang: the wanted language (defaults to *settings.PAGE_DEFAULT_LANGUAGE*)

pages.templatetags.pages_tags.**show_content**(*context*, *page*, *content_type*, *lang=None*, *fallback=True*)

Display a content type from a page.

Example:

```
{% show_content page_object "title" %}
```

You can also use the slug of a page:

```
{% show_content "my-page-slug" "title" %}
```

Or even the id of a page:

```
{% show_content 10 "title" %}
```

Parameters

- **page** – the page object, slug or id
- **content_type** – content_type used by a placeholder
- **lang** – the wanted language (default None, use the request object to know)
- **fallback** – use fallback content from other language

pages.templatetags.pages_tags.**show_revisions**(*context*, *page*, *content_type*, *lang=None*)

Render the last 10 revisions of a page content with a list using the `pages/revisions.html` template

1.13.4 Widgets

Django CMS come with a set of ready to use widgets that you can enable in the admin via a placeholder tag in your template.

class pages.widgets.**FileInput**(*page=None*, *language=None*, *attrs=None*, ***kwargs*)

delete_msg = 'Delete file'

property media

please_save_msg = 'Please save the page to show the file field'

render(*name*, *value*, *attrs=None*, ***kwargs*)

Render the widget as an HTML string.

class pages.widgets.**ImageInput**(*page=None*, *language=None*, *attrs=None*, ***kwargs*)

delete_msg = 'Delete image'

property media

please_save_msg = 'Please save the page to show the image field'

class pages.widgets.**LanguageChoiceWidget**(*language=None*, *attrs=None*, ***kwargs*)

property media

render(*name*, *value*, *attrs=None*, ***kwargs*)

Render the widget as an HTML string.

class pages.widgets.**PageLinkWidget**(*attrs=None*, *page=None*, *language=None*, *video_url=None*, *linkedpage=None*, *text=None*)

A page link *Widget* for the admin.

decompress(*value*)

Return a list of decompressed values for the given compressed value. The given value can be assumed to be valid, but not necessarily non-empty.

format_output(*rendered_widgets*)

Given a list of rendered widgets (as strings), it inserts an HTML linebreak between them.

Returns a Unicode string representing the HTML for the whole lot.

property media

Media for a multiwidget is the combination of all media of the subwidgets.

value_from_datadict(*data*, *files*, *name*)

Given a dictionary of data and this widget's name, return the value of this widget or None if it's not provided.

class pages.widgets.**RichTextarea**(*language=None*, *attrs=None*, ***kwargs*)

A RichTextarea widget.

class Media

```
css = {'all':  ['/static/pages/css/rte.css',
'/static/pages/css/font-awesome.min.css']}

js = ['/static/pages/javascript/jquery.js',
'/static/pages/javascript/jquery.rte.js']
```

property media

render(*name*, *value*, *attrs=None*, ***kwargs*)

Render the widget as an HTML string.

1.13.5 Page Model

class pages.models.**Page**(**args*, ***kwargs*)

This model contain the status, dates, author, template. The real content of the page can be found in the *Content* model.

creation_date

When the page has been created.

publication_date

When the page should be visible.

publication_end_date

When the publication of this page end.

last_modification_date
Last time this page has been modified.

status
The current status of the page. Could be DRAFT, PUBLISHED,
EXPIRED or HIDDEN. You should the property :attr:`calculated_status` if
you want that the dates are taken in account.

template
A string containing the name of the template file for this page.

exception DoesNotExist

exception MultipleObjectsReturned

property calculated_status

Get the calculated status of the page based on *Page.publication_date*, *Page.publication_end_date*, and *Page.status*.

content_by_language(*language*)

Return a list of latest published *Content* for a particluar language.

Parameters
language – wanted language,

expose_content()

Return all the current content of this page into a *string*.

This is used by the haystack framework to build the search index.

get_absolute_url(*language=None*)

Alias for *get_url_path*.

Parameters
language – the wanted url language.

get_children()

Cache superclass result

get_children_for_frontend()

Return a QuerySet of published children page

get_complete_slug(*language=None*, *hideroot=True*)

Return the complete slug of this page by concatenating all parent's slugs.

Parameters
language – the wanted slug language.

get_content(*language*, *ctype*, *language_fallback=False*)

Shortcut method for retrieving a piece of page content

Parameters

- **language** – wanted language, if not defined default is used.
- **ctype** – the type of content.
- **fallback** – if True, the content will also be searched in other languages.

get_date_ordered_children_for_frontend()

Return a QuerySet of published children page ordered by publication date.

get_languages()

Return a list of all used languages for this page.

get_template()

Get the *template* of this page if defined or the closer parent's one if defined or pages.settings.PAGE_DEFAULT_TEMPLATE otherwise.

get_template_name()

Get the template name of this page if defined or if a closer parent has a defined template or pages.settings.PAGE_DEFAULT_TEMPLATE otherwise.

get_url_path(*language=None*)

Return the URL's path component. Add the language prefix if PAGE_USE_LANGUAGE_PREFIX setting is set to True.

Parameters
language – the wanted url language.

invalidate()

Invalidate cached data for this page.

is_first_root()

Return True if this page is the first root pages.

margin_level()

Used in the admin menu to create the left margin.

move_to(*target*, *position='first-child'*)

Invalidate cache when moving

published_children()

Return a QuerySet of published children page

save(**args*, ***kwargs*)

Override the default save method.

slug(*language=None*, *fallback=True*)

Return the slug of the page depending on the given language.

Parameters

- **language** – wanted language, if not defined default is used.
- **fallback** – if True, the slug will also be searched in other languages.

slug_with_level(*language=None*)

Display the slug of the page prepended with insecable spaces to simluate the level of page in the hierarchy.

title(*language=None*, *fallback=True*)

Return the title of the page depending on the given language.

Parameters

- **language** – wanted language, if not defined default is used.
- **fallback** – if True, the slug will also be searched in other languages.

valid_targets()

Return a QuerySet of valid targets for moving a page into the tree.

Parameters
perms – the level of permission of the concerned user.

property visible

Return True if the page is visible on the frontend.

1.13.6 Page Manager

class pages.managers.**PageManager**(**args*, ***kwargs*)

Page manager provide several filters to obtain pages QuerySet that respect the page attributes and project settings.

drafts()

Creates a QuerySet of drafts using the page's Page.publication_date.

expired()

Creates a QuerySet of expired using the page's Page.publication_end_date.

filter_published(*queryset*)

Filter the given pages QuerySet to obtain only published page.

from_path(*complete_path*, *lang*, *exclude_drafts=True*)

Return a *Page* according to the page's path.

from_slug(*slug*)

hidden()

Creates a QuerySet of the hidden pages.

navigation()

Creates a QuerySet of the published root pages.

on_site(*site_id=None*)

Return a QuerySet of pages that are published on the site defined by the SITE_ID setting.

Parameters
site_id – specify the id of the site object to filter with.

published()

Creates a QuerySet of published *Page*.

root()

Return a QuerySet of pages without parent.

1.13.7 Page view

class pages.views.**Details**

This class based view get the root pages for navigation and the current page to display if there is any.

All is rendered with the current page's template.

choose_language(*lang*, *request*)

Deal with the multiple corner case of choosing the language.

extra_context(*request*, *context*)

Call the PAGE_EXTRA_CONTEXT function if there is one.

get_navigation(*request*, *path*, *lang*)

Get the pages that are at the root level.

get_template(*request*, *context*)

Just there in case you have special business logic.

is_user_staff(*request*)

Return True if the user is staff.

resolve_page(*request*, *context*, *is_staff*)

Return the appropriate page according to the path.

resolve_redirection(*request*, *context*)

Check for redirections.

1.13.8 Content Model

class pages.models.**Content**(**args*, ***kwargs*)

A block of content, tied to a *Page*, for a particular language

exception DoesNotExist

exception MultipleObjectsReturned

body

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

creation_date

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

get_next_by_creation_date(***, *field=<django.db.models.fields.DateTimeField: creation_date>*, *is_next=True*, ***kwargs*)

get_previous_by_creation_date(***, *field=<django.db.models.fields.DateTimeField: creation_date>*, *is_next=False*, ***kwargs*)

id

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

language

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

objects = <pages.managers.ContentManager object>

page

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

page_id

type

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

1.13.9 Content Manager

class pages.managers.**ContentManager**(**args*, ***kwargs*)

Content manager methods

PAGE_CONTENT_DICT_KEY = 'page_content_dict_%d_%s_%d'

create_content_if_changed(*page*, *language*, *ctype*, *body*)

Create a *Content* for a particular page and language only if the content has changed from the last time.

Parameters

- **page** – the concerned page object.
- **language** – the wanted language.
- **ctype** – the content type.
- **body** – the content of the Content object.

get_content(*page*, *language*, *ctype*, *language_fallback=False*)

Gets the latest content string for a particular page, language and placeholder.

Parameters

- **page** – the concerned page object.
- **language** – the wanted language.
- **ctype** – the content type.
- **language_fallback** – fallback to another language if `True`.

get_content_object(*page*, *language*, *ctype*)

Gets the latest published *Content* for a particular page, language and placeholder type.

get_content_slug_by_slug(*slug*)

Returns the latest *Content* slug object that match the given slug for the current site domain.

Parameters
slug – the wanted slug.

get_page_ids_by_slug(*slug*)

Return all page's id matching the given slug. This function also returns pages that have an old slug that match.

Parameters
slug – the wanted slug.

set_or_create_content(*page*, *language*, *ctype*, *body*)

Set or create a *Content* for a particular page and language.

Parameters

- **page** – the concerned page object.
- **language** – the wanted language.
- **ctype** – the content type.
- **body** – the content of the Content object.

1.13.10 PageAlias Model

class pages.models.**PageAlias**(**args*, ***kwargs*)

URL alias for a *Page*

exception DoesNotExist

exception MultipleObjectsReturned

id

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

objects = <pages.managers.PageAliasManager object>

page

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

page_id

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.

url

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

1.13.11 PageAlias Manager

class pages.managers.**PageAliasManager**(**args*, ***kwargs*)

PageAlias manager.

from_path(*request*, *path*, *lang*)

Resolve a request to an alias. returns a *PageAlias* if the url matches no page at all. The aliasing system supports plain aliases (`/foo/bar`) as well as aliases containing GET parameters (like `index.php?page=foo`).

Parameters

- **request** – the request object
- **path** – the complete path to the page
- **lang** – not used

1.13.12 Utils

A collection of functions for Page CMS

pages.utils.**get_now**()

pages.utils.**get_placeholders**(*template_name*)

Return a list of PlaceholderNode found in the given template.

Parameters

template_name – the name of the template file

pages.utils.**normalize_url**(*url*)

Return a normalized url with trailing and without leading slash.

```
>>> normalize_url(None)
'/'
>>> normalize_url('/')
'/'
>>> normalize_url('/foo/bar')
'/foo/bar'
>>> normalize_url('foo/bar')
'/foo/bar'
>>> normalize_url('/foo/bar/')
'/foo/bar'
```

pages.utils.**slugify**(*value*, *allow_unicode=False*)

Convert to ASCII if 'allow_unicode' is False. Convert spaces to hyphens. Remove characters that aren't alphanumerics, underscores, or hyphens. Convert to lowercase. Also strip leading and trailing whitespace. Copyright: https://docs.djangoproject.com/en/1.9/_modules/django/utils/text/#slugify TODO: replace after stopping support for Django 1.8

1.13.13 Http

Page CMS functions related to the `request` object.

`pages.phttp.`**`get_language_from_request`**(*request*)

Return the most obvious language according the request.

`pages.phttp.`**`get_request_mock`**()

Build a `request` mock up for tests

`pages.phttp.`**`get_slug`**(*path*)

Return the page's slug

```
>>> get_slug('/test/function/')
function
```

`pages.phttp.`**`get_template_from_request`**(*request*, *page=None*)

Gets a valid template from different sources or falls back to the default template.

`pages.phttp.`**`remove_slug`**(*path*)

Return the remainin part of the path

```
>>> remove_slug('/test/some/function/')
test/some
```

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

H

I

J

L

M

N

O

P

R

S

T

U

V

W