

---

# **django cms Documentation**

*Release 2.1.5*

**Patrick Lauber**

May 27, 2016



<b>1</b>	<b>Getting Started</b>	<b>3</b>
1.1	Installation . . . . .	3
1.2	Introductory Tutorial . . . . .	4
1.3	Using South with Django-CMS . . . . .	11
1.4	Configuration . . . . .	11
1.5	Navigation . . . . .	18
1.6	Plugins reference . . . . .	20
<b>2</b>	<b>Advanced</b>	<b>25</b>
2.1	Internationalization . . . . .	25
2.2	Sitemap Guide . . . . .	26
2.3	Templatetags . . . . .	26
<b>3</b>	<b>Extending the CMS</b>	<b>33</b>
3.1	Extending the CMS: Examples . . . . .	33
3.2	Custom Plugins . . . . .	37
3.3	App Integration . . . . .	42
3.4	API References . . . . .	47
3.5	Placeholders outside the CMS . . . . .	49
3.6	Search and the Django-CMS . . . . .	50
<b>4</b>	<b>Contributing to the CMS</b>	<b>53</b>
4.1	Contributing to Django-CMS . . . . .	53
4.2	Indices and tables . . . . .	55
	<b>Python Module Index</b>	<b>57</b>



This document refers to version 2.1.5



---

## Getting Started

---

### 1.1 Installation

This document assumes you are familiar with Python and Django, and should outline the steps necessary for you to follow the [Introductory Tutorial](#).

#### 1.1.1 Requirements

- Python 2.5 (or a higher release of 2.x).
- Django 1.2.3 (or a higher release of 1.2).
- South 0.7.2 or higher
- PIL 1.1.6 or higher
- django-classy-tags 0.2.2 or higher
- An installed and working instance of one of the databases listed in the [Databases](#) section.

---

**Note:** When installing the django CMS using pip, both Django and django-classy-tags will be installed automatically.

---

#### On Ubuntu

If you're using Ubuntu (tested with 10.10), the following should get you started:

```
sudo aptitude install python2.6 python-setuptools python-imaging
sudo easy_install pip
sudo pip install django-cms south django-appmedia
```

Additionally, you need the python driver for your selected database:

```
sudo aptitude python-psycopg2 or sudo aptitude install python-mysql
```

This will install PIL and your database's driver globally.

You have now everything that is needed for you to follow the [Introductory Tutorial](#).

#### On Mac OSX

**TODO** (Should setup everything up to but not including “pip install django-cms” like the above)

## On Microsoft Windows

TODO.

### 1.1.2 Databases

We recommend using [PostgreSQL](#) or [MySQL](#) with Django CMS. Installing and maintaining database systems is outside the scope of this documentation, but is very well documented on the system's respective websites.

To use Django CMS efficiently, we recommend:

- Create a separate set of credentials for django CMS.
- Create a separate database for django CMS to use.

## 1.2 Introductory Tutorial

This guide assumes your machine meets the requirements outlined in the [Installation](#) section of this documentation.

### 1.2.1 Configuration and setup

#### Preparing the environment

Gathering the requirements is a good start, but we now need to give the CMS a Django project to live in, and configure it.

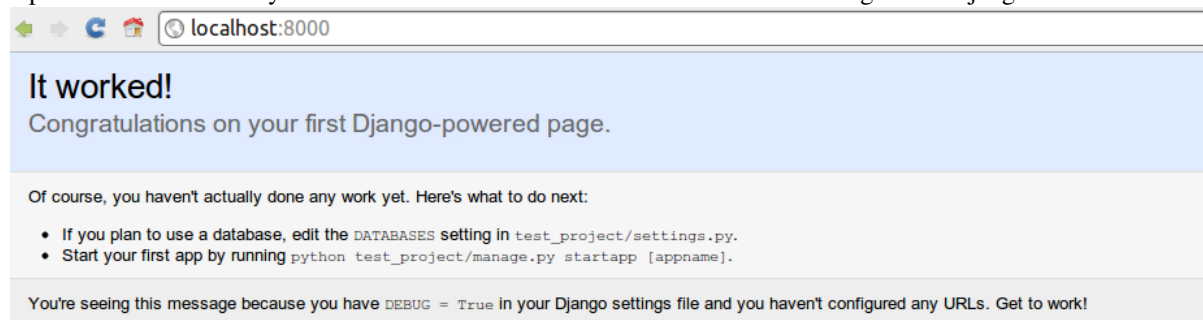
#### Starting your Django project

The following assumes your project will be in `~/workspace/myproject/`.

Set up your Django project:

```
cd ~/workspace
django-admin.py startproject myproject
cd myproject
python manage.py runserver
```

Open `127.0.0.1:8000` in your browser. You should see a nice “It Worked” message from Django.



#### Installing and configuring django CMS in your Django project

Open the file `~/workspace/myproject/settings.py`.

To make your life easier, add the following at the top of the file:



```
# -*- coding: utf-8 -*-
import os
gettext = lambda s: s
PROJECT_PATH = os.path.abspath(os.path.dirname(__file__))
```

Add the following apps to your `INSTALLED_APPS`:

- 'cms'
- 'mptt'
- 'menus'
- 'south'
- 'appmedia'

Also add any (or all) of the following plugins, depending on your needs:

- 'cms.plugins.text'
- 'cms.plugins.picture'
- 'cms.plugins.link'
- 'cms.plugins.file'
- 'cms.plugins.snippet'
- 'cms.plugins.googlemap'

If you wish to use the moderation workflow, also add:

- 'publisher'

Further, make sure you uncomment 'django.contrib.admin'

You need to add the django CMS middlewares to your `MIDDLEWARE_CLASSES` at the right position:

```
MIDDLEWARE_CLASSES = (
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'cms.middleware.page.CurrentPageMiddleware',
    'cms.middleware.user.CurrentUserMiddleware',
    'cms.middleware.toolbar.ToolbarMiddleware',
    'cms.middleware.media.PlaceholderMediaMiddleware',
)
```

You need at least the following `TEMPLATE_CONTEXT_PROCESSORS` (a default Django settings file will not have any):

```
TEMPLATE_CONTEXT_PROCESSORS = (
    'django.core.context_processors.auth',
    'django.core.context_processors.i18n',
    'django.core.context_processors.request',
    'django.core.context_processors.media',
    'cms.context_processors.media',
)
```

Almost there! Point your `MEDIA_ROOT` to where the static media should live (that is, your images, CSS files, Javascript files...):

```
MEDIA_ROOT = os.path.join(PROJECT_PATH, "media")
MEDIA_URL = "/media/"
ADMIN_MEDIA_PREFIX="/media/admin/"
```

Now add a little magic to the `TEMPLATE_DIRS` section of the file:

```
TEMPLATE_DIRS = (  
    # The docs say it should be absolute path: PROJECT_PATH is precisely one.  
    # Life is wonderful!  
    os.path.join(PROJECT_PATH, "templates")  
)
```

Add at least one template to `CMS_TEMPLATES`; for example:

```
CMS_TEMPLATES = (  
    ('template_1.html', 'Template One'),  
    ('template_2.html', 'Template Two'),  
)
```

We will create the actual template files at a later step, don't worry about it for now, and simply paste this code in your settings file.

---

**Note:** The templates you define in `CMS_TEMPLATES` have to exist at runtime and contain at least one `{% placeholder <name> %}` template tag to be useful for django CMS. For more details see [Creating templates](#)

---

The django CMS will allow you to edit all languages which Django has built in translations for, this is way too many so we'll limit it to English for now:

```
LANGUAGES = [  
    ('en', 'English'),  
]
```

Finally, setup the `DATABASES` part of the file to reflect your database deployment. If you just want to try out things locally, `sqlite3` is the easiest database to set up, however it should not be used in production. If you still wish to use it for now, this is what your `DATABASES` setting should look like:

```
DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.sqlite3',  
        'NAME': os.path.join(PROJECT_DIR, 'database.sqlite'),  
    }  
}
```

## URL configuration

You need to include the `'cms.urls'` urlpatterns **at the end** of your urlpatterns. We suggest starting with the following `urls.py`:

```
from django.conf.urls.defaults import *  
from django.contrib import admin  
from django.conf import settings  
  
admin.autodiscover()  
  
urlpatterns = patterns('',  
    (r'^admin/', include(admin.site.urls)),  
    url(r'^$', include('cms.urls')),  
)  
  
if settings.DEBUG:  
    urlpatterns = patterns('',  
        (r'^' + settings.MEDIA_URL.lstrip('/'), include('appmedia.urls')),  
    ) + urlpatterns
```

To have access to app specific media files, use `python manage.py symlinkmedia` and `django-appmedia` will do all the work for you.

## Initial database setup

This command depends on whether you **upgrade** your installation or do a **fresh install**. We recommend that you get familiar with the way [South](#) works, as it is a very powerful, easy and convenient tool. Django CMS uses it extensively.

### Fresh install

Run:

```
python manage.py syncdb --all
python manage.py migrate --fake
```

The first command will prompt you to create a super user; choose ‘yes’ and enter appropriate values.

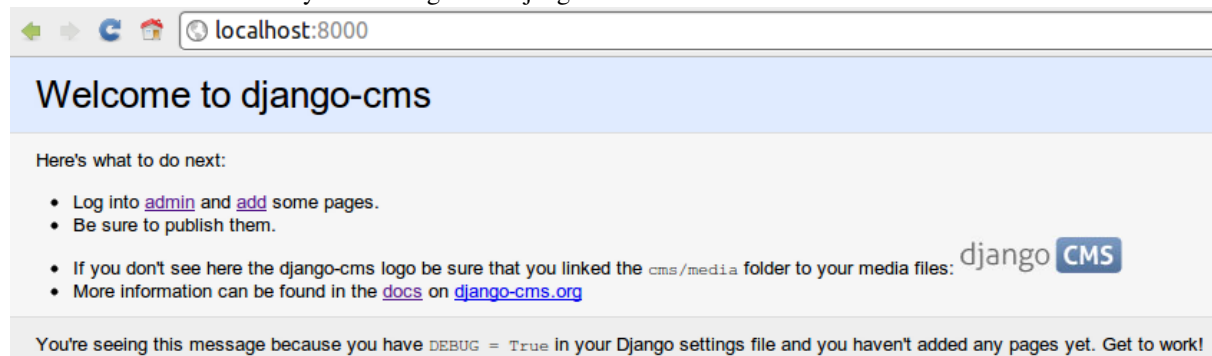
### Upgrade

Run:

```
python manage.py syncdb
python manage.py migrate
```

## Up and running!

That should be it. Restart your development server using `python manage.py runserver` and point a web browser to `127.0.0.1:8000`: you should get the Django CMS “It Worked” screen.



Head over to the *admin panel* `<http://127.0.0.1:8000/admin/>` and log in with the user you created during the database setup.

To deploy your django CMS project on a production webserver, please refer to the [Django Documentation](#).

### 1.2.2 Creating templates

Django CMS uses templates to define how a page should look and what parts of it are editable. Editable areas are called *placeholders*. These templates are standard Django templates and you may use them as described in the [official documentation](#).

Templates you wish to use on your pages must be declared in the `CMS_TEMPLATES` setting:

```
CMS_TEMPLATES = (
    ('template_1.html', 'Template One'),
    ('template_2.html', 'Template Two'),
)
```

If you followed this tutorial from the beginning, we already put this code in your settings file.

Now, on with the actual template files!

Fire up your favorite editor and create a file called `base.html` in a folder called `templates` in your `myproject` directory.

Here is a simple example for a base template called `base.html`:

```
{% load cms_tags %}
<html>
  <body>
    {% placeholder base_content %}
    {% block base_content %}{% endblock %}
  </body>
</html>
```

Now, create a file called `template_1.html` in the same directory. This will use your base template, and add extra content to it:

```
{% extends "base.html" %}
{% load cms_tags %}

{% block base_content %}
  {% placeholder template_1_content %}
{% endblock %}
```

When you set `template_1.html` as a template on a page you will get two placeholders to put plugins in. One is `template_1_content` from the page template `template_1.html` and another is `base_content` from the extended `base.html`.

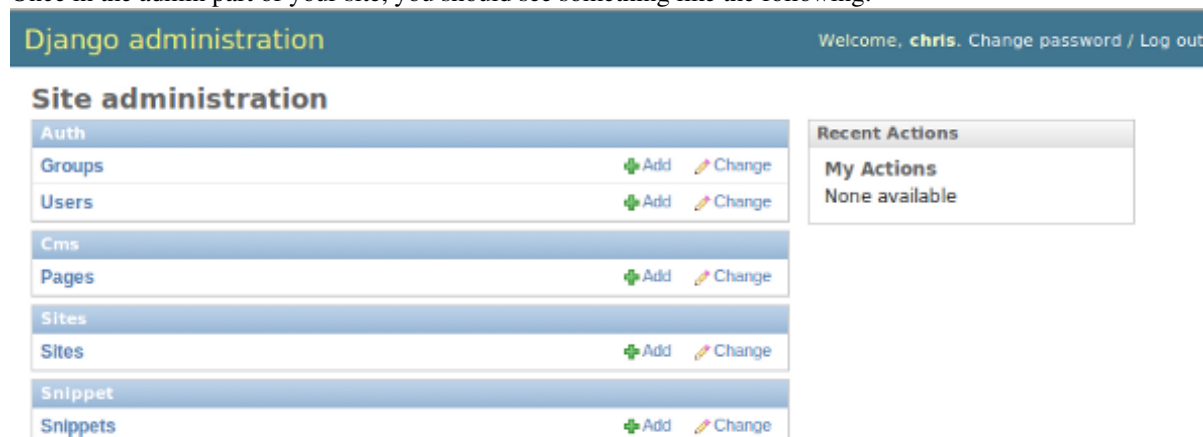
When working with a lot of placeholders, make sure to give descriptive names for your placeholders, to more easily identify them in the admin panel.

Now, feel free to experiment and make a `template_2.html` file! If you don't feel creative, just copy `template_1` and name the second placeholder something like "template\_2\_content".

### 1.2.3 Creating your first CMS page!

That's it, now the best part: you can start using the CMS! Run your server with `python manage.py runserver`, then point a web browser to `127.0.0.1:8000/admin/`, and log in using the super user credentials you defined when you ran `syncdb` earlier.

Once in the admin part of your site, you should see something like the following:

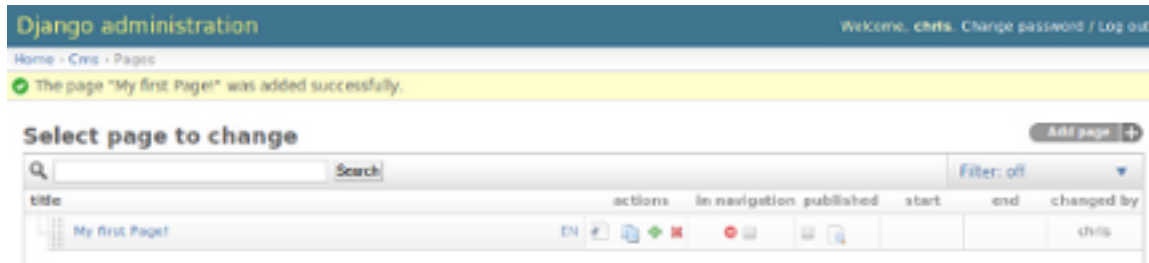


## Adding a page

Adding a page is as simple as clicking “Pages” in the admin view, then the “add page” button on the top right-hand corner of the screen.

This is where you select which template to use (remember, we created two), as well as pretty obvious things like which language the page is in (used for internationalisation), the page’s title, and the url slug it will use.

Hitting the “Save” button, well, saves the page. It will now display in the list of pages.



Congratulations! You now have a fully functional Django CMS installation!

## Publishing a page

The list of pages available is a handy way to change a few parameters about your pages:

### Visibility

By default, pages are “invisible”. To let people access them you should mark them as “published”.

### Menus

Another option this view lets you tweak is whether or not the page should appear in your site’s navigation (that is, whether there should be a menu entry to reach it or not)

## Adding content to a page

So far, our page doesn’t do much. Make sure it’s marked as “published”, then click on the page’s “edit” button.

Ignore most of the interface for now, and click the “view on site” button on the top right-hand corner of the screen. As expected, your page is blank for the time being, since our template is really a minimal one.

Let’s get to it now then!

Press your browser’s back button, so as to see the page’s admin interface. If you followed the tutorial so far, your template (`template_1.html`) defines two placeholders. The admin interface shows you these placeholders as sub menus:

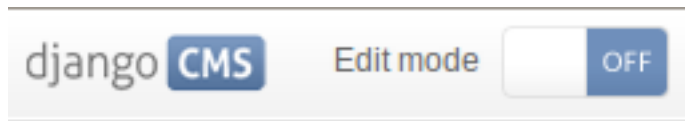


Scroll down the “Available plugins” drop-down list. This displays the plugins you added to your `INSTALLED_APPS` settings. Choose the “text” plugin in the drop-down, then press the “Add” button.

The right part of the plugin area displays a rich text editor (TinyMCE).

Type in whatever you please there, then press the “Save” button.

Go back to your website using the top right-hand “View on site” button. That’s it!



## Hello, CMS world!

### Where to go from here

Congratulations, you now have a fully functional CMS! Feel free to play around with the different plugins provided out of the box, and build great websites!

### 1.2.4 Troubleshooting

If you’ve created a page & you don’t see it in the cms list of the Django admin:

- Be sure you copied all the media files. Check with firebug and its “net” panel to see if you have any 404s.

If you’re editing a Page in the Django admin, but don’t see an “Add Plugin” button with a dropdown-list of plugins:

- Be sure your `CMS_TEMPLATES` setting is correct, the templates specified exist, and they contain at least one `{% placeholder %} templatetag`.

### Template errors

If your placeholder content isn’t displayed when you view a CMS page: change the `CMS_MODERATOR` variable in `settings.py` to `False`. This bug has been recently fixed, so upgrade to the latest version of Django CMS. See: <https://github.com/divio/django-cms/issues/issue/430>

### Javascript errors

If plugins don’t work (e.g.: you add a text plugin, but don’t see the Javascript text editor in the plugin window), you should use a Javascript inspector in your browser to investigate the issue (e.g.: Firebug for Firefox, Web Inspector for Safari or Chrome). The Javascript inspector may report the following errors:

- **TypeError: Result of expression ‘jQuery’ [undefined] is not a function.**

If you see this, check the `MEDIA_URL` variable in your `settings.py` file. Your webserver (e.g.: Apache) should be configured to serve static media files from this URL.

- **Unsafe JavaScript attempt to access frame with URL `http://localhost/media/cms/wymeditor/iframe/default/wymiframe` from frame with URL `http://127.0.0.1:8000/admin/cms/page/1/edit-plugin/2/`. Domains, protocols and ports must match.**

This error is due to the Django test server running on a different port and URL than the main webserver. In your test environment, you can overcome this issue by adding a `CMS_MEDIA_URL` variable to your `settings.py` file, and adding a url rule in `urls.py` to make the Django development serve the Django CMS files from this location.

## 1.3 Using South with Django-CMS

South is an incredible piece of software that lets you handle database migrations. This document is by no means meant to replace the excellent [documentation](#) available online, but rather to give a quick primer on how and why to get started quickly with South.

### 1.3.1 Installation

Using Django and Python is, as usual, a joy. Installing South should mostly be as easy as typing:

```
pip install South
```

Then, simply add “South” to the list of `INSTALLED_APPS` in your `settings.py` file.

### 1.3.2 Basic usage

For a very short crash course:

1. Instead of the initial `manage.py syncdb` command, simply run `manage.py schemamigration --initial <app name>`. This will create a new migrations package, along with a new migration file (in the form of a python script).
2. Run the migration using `manage.py migrate`. Your tables have now been created in the database, Django will work as usual
3. Whenever you make changes to your `models.py` file, run `manage.py schemamigration --auto <app name>` to create a new migration file, then `manage.py migrate` to apply the newly created migration!

### 1.3.3 More information about South

Obviously, South is a very powerful tool and this simple crash course is only the very tip of the iceberg. Readers are highly encouraged to have a quick glance at the excellent official South [documentation](#).

## 1.4 Configuration

The Django-CMS has a lot of settings you can use to customize your installation of the CMS to be exactly like you want it to be.

### 1.4.1 Required Settings

#### CMS\_TEMPLATES

Default: None (Not a valid setting!)

A list of templates you can select for a page.

Example:

```
CMS_TEMPLATES = (  
    ('base.html', gettext('default')),  
    ('2col.html', gettext('2 Column')),  
    ('3col.html', gettext('3 Column')),  
    ('extra.html', gettext('Some extra fancy template')),  
)
```

## 1.4.2 Basic Customization

### CMS\_TEMPLATE\_INHERITANCE

Default: True

*Optional* Enables the inheritance of templates from parent pages.

If this is enabled, pages have the additional template option to inherit their template from the nearest ancestor. New pages default to this setting if the new page is not a root page.

### CMS\_PLACEHOLDER\_CONF

Default: {} **Optional**

Used to configure placeholders. If not given, all plugins are available in all placeholders.

Example:

```
CMS_PLACEHOLDER_CONF = {
    'content': {
        'plugins': ('TextPlugin', 'PicturePlugin'),
        'text_only_plugins': ('LinkPlugin',),
        'extra_context': {"width":640},
        'name':gettext("Content"),
    },
    'right-column': {
        "plugins": ('TeaserPlugin', 'LinkPlugin'),
        "extra_context": {"width":280},
        'name':gettext("Right Column"),
        'limits': {
            'global': 2,
            'TeaserPlugin': 1,
            'LinkPlugin': 1,
        },
    },
    'base.html content': {
        "plugins": ('TextPlugin', 'PicturePlugin', 'TeaserPlugin')
    },
}
```

You can combine template names and placeholder names to granually define plugins, as shown above with ‘base.html content’.

#### **plugins**

A list of plugins that can be added to this placeholder. If not supplied, all plugins can be selected.

#### **text\_only\_plugins**

A list of additional plugins available only in the TextPlugin, these plugins can’t be added directly to this placeholder.

#### **extra\_context**

Extra context that plugins in this placeholder receive.

#### **name**

The name displayed in the Django admin. With the gettext stub, the name can be internationalized.

#### **limits**

Limit the number of plugins that can be placed inside this placeholder. Dictionary keys are plugin names; values are their respective limits. Special case: “global” - Limit the absolute number of plugins in this placeholder regardless of type (takes precedence over the type-specific limits).



## CMS\_PLUGIN\_CONTEXT\_PROCESSORS

Default: []

A list of plugin context processors. Plugin context processors are callables that modify all plugin's context before rendering. See [Custom Plugins](#) for more information.

## CMS\_PLUGIN\_PROCESSORS

Default: []

A list of plugin processors. Plugin processors are callables that modify all plugin's output after rendering. See [Custom Plugins](#) for more information.

## CMS\_APPHOOKS

Default: ()

A list of import paths for `cms.app_base.CMSApp` subclasses.

Defaults to an empty list which means CMS applications are auto-discovered in all `INSTALLED_APPS` by trying to import their `cms_app` module.

If this setting is set, the auto-discovery is disabled.

Example:

```
CMS_APPHOOKS = (
    'myapp.cms_app.MyApp',
    'otherapp.cms_app.MyFancyApp',
    'sampleapp.cms_app.SampleApp',
)
```

## PLACEHOLDER\_FRONTEND\_EDITING

Default: True

If set to False, frontend editing is not available for models using `cms.models.fields.PlaceholderField`.

### 1.4.3 I18N and L10N

#### CMS\_HIDE\_UNTRANSLATED

Default: True

By default django-cms hides menu items that are not yet translated into the current language. With this setting set to False they will show up anyway.

#### CMS\_LANGUAGES

Default: Value of `LANGUAGES`

Defines the languages available in the CMS.

Example:

```
CMS_LANGUAGES = (  
    ('fr', gettext('French')),  
    ('de', gettext('German')),  
    ('en', gettext('English')),  
)
```

---

**Note:** Make sure you only define languages which are also in `LANGUAGES`.

---

### **CMS\_LANGUAGE\_FALLBACK**

Default: `True`

This will redirect the browser to the same page in another language if the page is not available in the current language.

### **CMS\_LANGUAGE\_CONF**

Default: `{}`

Language fallback ordering for each language.

Example:

```
CMS_LANGUAGE_CONF = {  
    'de': ['en', 'fr'],  
    'en': ['de'],  
}
```

### **CMS\_SITE\_LANGUAGES**

Default: `{}`

If you have more than one site and `CMS_LANGUAGES` differs between the sites, you may want to fill this out so if you switch between the sites in the admin you only get the languages available on this site.

Example:

```
CMS_SITE_LANGUAGES = {  
    1: ['en', 'de'],  
    2: ['en', 'fr'],  
    3: ['en'],  
}
```

### **CMS\_FRONTEND\_LANGUAGES**

Default: Value of `CMS_LANGUAGES`

A list of languages Django CMS uses in the frontend. For example, if you decide you want to add a new language to your page but don't want to show it to the world yet.

Example:

```
CMS_FRONTEND_LANGUAGES = ("de", "en", "pt-BR")
```

## CMS\_DBGETTEXT

Default: `False` (unless `dbgettext` is in `settings.INSTALLED_APPS`)

Enable gettext-based translation of CMS content rather than use the standard administration interface. Requires `django-dbgettext`.

**Warning:** This feature is deprecated and will be removed in 2.2.

## CMS\_DBGETTEXT\_SLUGS

Default: `False`

Enable gettext-based translation of page paths/slugs. Experimental at this stage, as resulting translations cannot be guaranteed to be unique.

For general `dbgettext` settings, see the [dbgettext documentation](#).

**Warning:** This feature is deprecated and will be removed in 2.2.

## 1.4.4 Media Settings

### CMS\_MEDIA\_PATH

default: `cms/`

The path from `MEDIA_ROOT` to the media files located in `cms/media/`

### CMS\_MEDIA\_ROOT

Default: `settings.MEDIA_ROOT + CMS_MEDIA_PATH`

The path to the media root of the cms media files.

### CMS\_MEDIA\_URL

default: `MEDIA_URL + CMS_MEDIA_PATH`

The location of the media files that are located in `cms/media/cms/`

### CMS\_PAGE\_MEDIA\_PATH

Default: `'cms_page_media/'`

By default, Django CMS creates a folder called `'cms_page_media'` in your static files folder where all uploaded media files are stored. The media files are stored in subfolders numbered with the id of the page.

## 1.4.5 URLs

### CMS\_URL\_OVERWRITE

Default: `True`

This adds a new field “url overwrite” to the “advanced settings” tab of your page. With this field you can overwrite the whole relative url of the page.

## CMS\_MENU\_TITLE\_OVERWRITE

Default: `False`

This adds a new “menu title” field beside the title field.

With this field you can overwrite the title that is displayed in the menu.

To access the menu title in the template, use:

```
{{ page.get_menu_title }}
```

## CMS\_REDIRECTS

Default: `False`

This adds a new “redirect” field to the “advanced settings” tab of the page

You can set a url here, which a visitor will be redirected to when the page is accessed.

Note: Don’t use this too much. `django.contrib.redirect` is much more flexible, handy, and is designed exactly for this purpose.

## CMS\_FLAT\_URLS

Default: `False`

If this is enabled the slugs are not nested in the urls.

So a page with a “world” slug will have a “/world” url, even it is a child of the “hello” page. If disabled the page would have the url: “/hello/world/”

## CMS\_SOFTROOT

Default: `False`

This adds a new “softroot” field to the “advanced settings” tab of the page. If a page is marked as softroot the menu will only display items until it finds the softroot.

If you have a huge site you can easily partition the menu with this.

## 1.4.6 Advanced Settings

### CMS\_PERMISSION

Default: `False`

If this is enabled you get 3 new models in Admin:

- Pages global permissions
- User groups - page
- Users - page

In the edit-view of the pages you can now assign users to pages and grant them permissions. In the global permissions you can set the permissions for users globally.

If a user has the right to create new users he can now do so in the “Users - page”. But he will only see the users he created. The users he created can also only inherit the rights he has. So if he only has been granted the right to edit a certain page all users he creates can, in turn, only edit this page. Naturally he can limit the rights of the users he creates even further, allowing them to see only a subset of the pages he’s allowed access to, for example.

## CMS\_MODERATOR

Default: `False`

If set to true, gives you a new “moderation” column in the tree view.

You can select to moderate pages or whole trees. If a page is under moderation you will receive an email if somebody changes a page and you will be asked to approve the changes. Only after you approved the changes will they be updated on the “live” site. If you make changes to a page you moderate yourself, you will need to approve it anyway. This allows you to change a lot of pages for a new version of the site, for example, and go live with all the changes at the same time.

## CMS\_SHOW\_START\_DATE & CMS\_SHOW\_END\_DATE

Default: `False` for both

This adds 2 new date-time fields in the advanced-settings tab of the page. With this option you can limit the time a page is published.

## CMS\_SEO\_FIELDS

Default: `False`

This adds a new “SEO Fields” fieldset to the page admin. You can set the Page Title, Meta Keywords and Meta Description in there.

To access these fields in the template use:

```
{% load cms_tags %}
<head>
  <title>{% page_attribute page_title %}</title>
  <meta name="description" content="{% page_attribute meta_description %}"/>
  <meta name="keywords" content="{% page_attribute meta_keywords %}"/>
  ...
  ...
</head>
```

## CMS\_CONTENT\_CACHE\_DURATION

Default: `60`

Cache expiration (in seconds) for `show_placeholder` and `page_url` template tags.

## MENU\_CACHE\_DURATION

Default: `3600`

Cache expiration (in seconds) for the menu tree.

## CMS\_CACHE\_PREFIX

Default: `None`

The CMS will prepend the value associated with this key to every cache access (set and get). This is useful when you have several Django-CMS installations, and you don’t want them to share cache objects.

Example:

```
CMS_CACHE_PREFIX = 'mysite-live'
```

## 1.5 Navigation

There are four template tags for use in the templates that are connected to the menu:

- `show_menu`
- `show_menu_below_id`
- `show_sub_menu`
- `show_breadcrumb`

---

**Note:** Please note that menus were originally implemented to be application-independant and as such, live in the `menus` application instead of the “normal” `cms`

---

### 1.5.1 `show_menu`

`{% show_menu %}` renders the navigation of the current page. You can overwrite the appearance and the HTML if you add a `menu/menu.html` template to your project or edit the one provided with `django-cms`. `show_menu` takes four optional parameters: `start_level`, `end_level`, `extra_inactive`, and `extra_active`.

The first two parameters, `start_level` (default=0) and `end_level` (default=100) specify from what level to which level should the navigation be rendered. If you have a home as a root node and don't want to display home you can render the navigation only after level 1.

The third parameter, `extra_inactive` (default=0), specifies how many levels of navigation should be displayed if a node is not a direct ancestor or descendant of the current active node.

Finally, the fourth parameter, `extra_active` (default=100), specifies how many levels of descendants of the currently active node should be displayed.

### Some Examples

Complete navigation (as a nested list):

```
{% load cache menu_tags %}
<ul>
    {% show_menu 0 100 100 100 %}
</ul>
```

Navigation with active tree (as a nested list):

```
<ul>
    {% show_menu 0 100 0 100 %}
</ul>
```

Navigation with only one active extra level:

```
<ul>
    {% show_menu 0 100 0 1 %}
</ul>
```

Level 1 navigation (as a nested list):

```
<ul>
    {% show_menu 1 %}
</ul>
```

Navigation with a custom template:

```
{% show_menu 0 100 100 100 "myapp/menu.html" %}
```

### 1.5.2 show\_menu\_below\_id

If you have set an id in the advanced settings of a page, you can display the submenu of this page with a template tag. For example, we have a page called meta that is not displayed in the navigation and that has the id “meta”:

```
<ul>
  {% show_menu_below_id "meta" %}
</ul>
```

You can give it the same optional parameters as show\_menu:

```
<ul>
  {% show_menu_below_id "meta" 0 100 100 100 "myapp/menu.html" %}
</ul>
```

### 1.5.3 show\_sub\_menu

Display the sub menu of the current page (as a nested list). Takes one argument that specifies how many levels deep should the submenu be displayed. The template can be found at menu/sub\_menu.html:

```
<ul>
  {% show_sub_menu 1 %}
</ul>
```

Or with a custom template:

```
<ul>
  {% show_sub_menu 1 "myapp/submenu.html" %}
</ul>
```

### 1.5.4 show\_breadcrumb

Show the breadcrumb navigation of the current page. The template for the HTML can be found at menu/breadcrumb.html:

```
{% show_breadcrumb %}
```

Or with a custom template and only display level 2 or higher:

```
{% show_breadcrumb 2 "myapp/breadcrumb.html" %}
```

If the current URL is not handled by the CMS or you are working in a navigation extender, you may need to provide your own breadcrumb via the template. This is mostly needed for pages like login, logout and third-party apps.

### 1.5.5 Properties of Navigation Nodes in templates

```
{{ node.is_leaf_node }}
```

Is it the last in the tree? If true it doesn't have any children. (This normally comes from mptt.)

```
{{ node.level }}
```

The level of the node. Starts at 0.

```
{{ node.menu_level }}
```

The level of the node from the root node of the menu. Starts at 0. If your menu starts at level 1 or you have a “soft root” (described in the next section) the first node still would have 0 as its *menu\_level*.

```
{{ node.get_absolute_url }}
```

The absolute URL of the node.

```
{{ node.get_title }}
```

The title in the current language of the node.

```
{{ node.selected }}
```

If true this node is the current one selected/active at this URL.

```
{{ node.ancestor }}
```

If true this node is an ancestor of the current selected node.

```
{{ node.sibling }}
```

If true this node is a sibling of the current selected node.

```
{{ node.descendant }}
```

If true this node is a descendant of the current selected node.

```
{{ node.soft_root }}
```

If true this node is a “soft root”.

### 1.5.6 Soft Roots

“Soft roots” are pages that start a new navigation. If you are in a child of a soft root node you can only see the path to the soft root. This feature is useful if you have big navigation trees with a lot of pages and don’t want to overwhelm the user.

To enable it put the following in your `settings.py` file:

```
CMS_SOFTROOT = True
```

Now you can mark a page as “soft root” in the ‘Advanced’ tab of the page’s settings in the admin interface.

### 1.5.7 Modifying & Extending the menu

Please refer to the [App Integration](#) documentation

## 1.6 Plugins reference

### 1.6.1 File

Allows you to upload a file. A filetype icon will be assigned based on the file extension.

For installation be sure you have the following in the `INSTALLED_APPS` setting in your project’s `settings.py` file:



```
INSTALLED_APPS = (  
    # ...  
    'cms.plugins.file',  
    # ...  
)
```

### 1.6.2 Flash

Allows you to upload and display a Flash SWF file on your page.

For installation be sure you have the following in the `INSTALLED_APPS` setting in your project's `settings.py` file:

```
INSTALLED_APPS = (  
    # ...  
    'cms.plugins.flash',  
    # ...  
)
```

### 1.6.3 GoogleMap

Displays a map of an address on your page.

For installation be sure you have the following in the `INSTALLED_APPS` setting in your project's `settings.py` file:

```
INSTALLED_APPS = (  
    # ...  
    'cms.plugins.googlemap',  
    # ...  
)
```

The Google Maps API key is also required. You can either put this in a project setting called `GOOGLE_MAPS_API_KEY` or be sure the template context has a variable with the same name.

### 1.6.4 Link

Displays a link to an arbitrary URL or to a page. If a page is moved the URL will still be correct.

For installation be sure to have the following in the `INSTALLED_APPS` setting in your project's `settings.py` file:

```
INSTALLED_APPS = (  
    # ...  
    'cms.plugins.link',  
    # ...  
)
```

### 1.6.5 Picture

Displays a picture in a page.

For installation be sure you have the following in the `INSTALLED_APPS` setting in your project's `settings.py` file:

```
INSTALLED_APPS = (  
    # ...  
    'cms.plugins.picture',  
)
```

```
)  
# ...
```

If you want to resize the picture you can get a thumbnail library. We recommend [sorl.thumbnail](#).

In your project template directory create a folder called `cms/plugins` and create a file called `picture.html` in there. Here is an example `picture.html` template:

```
{% load i18n thumbnail %}  
{% spaceless %}  
  
{% if picture.url %}<a href="{{ picture.url }}">{% endif %}  
{% ifequal placeholder "content" %}  
      
{% endifequal %}  
{% ifequal placeholder "teaser" %}  
      
{% endifequal %}  
{% if picture.url %}</a>{% endif %}  
  
{% endspaceless %}
```

In this template the picture is scaled differently based on which placeholder it was placed in.

### 1.6.6 Snippet

Just renders some HTML snippet. Mostly used for development or hackery.

For installation be sure you have the following in the `INSTALLED_APPS` setting in your project's `settings.py` file:

```
INSTALLED_APPS = (  
    # ...  
    'cms.plugins.snippet',  
    # ...  
)
```

### 1.6.7 Teaser

Displays a teaser box for another page or a URL. A picture and a description can be added.

For installation be sure you have the following in the `INSTALLED_APPS` settings in your project's `settings.py` file:

```
INSTALLED_APPS = (  
    # ...  
    'cms.plugins.teaser',  
    # ...  
)
```

### 1.6.8 Text

Displays text. If plugins are text-enabled they can be placed inside the text-flow. At this moment the following plugins are text-enabled:

- link
- picture
- file
- snippet

The current editor is [Wymeditor](#). If you want to use TinyMce you need to install [django-tinymce](#). If `tinymce` is in your `INSTALLED_APPS` it will be automatically enabled. If you have `tinymce` installed but don't want to use it in the cms put the following in your `settings.py`:

```
CMS_USE_TINYMCE = False
```

For installation be sure you have the following in your project's `INSTALLED_APPS` setting:

```
INSTALLED_APPS = (  
    # ...  
    'cms.plugins.text',  
    # ...  
)
```

## 1.6.9 Video

Plays Video Files or Youtube / Vimeo Videos. Uses the [OSFlashVideoPlayer](#). If you upload a file use `.flv` files or h264 encoded video files.

For installation be sure you have the following in your project's `INSTALLED_APPS` setting:

```
INSTALLED_APPS = (  
    # ...  
    'cms.plugins.video',  
    # ...  
)
```

There are some settings you can set in your `settings.py` to overwrite some default behavior:

- `VIDEO_AUTOPLAY` default=False
- `VIDEO_AUTOHIDE` default=False
- `VIDEO_FULLSCREEN` default=True
- `VIDEO_LOOP` default=False
- `VIDEO_AUTOPLAY` default=False
- `VIDEO_AUTOPLAY` default=False
- `VIDEO_BG_COLOR` default="000000"
- `VIDEO_TEXT_COLOR` default="FFFFFF"
- `VIDEO_SEEKBAR_COLOR` default="13ABEC"
- `VIDEO_SEEKBARBG_COLOR` default="333333"
- `VIDEO_LOADINGBAR_COLOR` default="828282"
- `VIDEO_BUTTON_OUT_COLOR` default="333333"
- `VIDEO_BUTTON_OVER_COLOR` default="000000"
- `VIDEO_BUTTON_HIGHLIGHT_COLOR` default="FFFFFF"

## 1.6.10 Twitter

Displays the last number of post of a twitter user.

For installation be sure you have the following in your project's `INSTALLED_APPS` setting:

```
INSTALLED_APPS = (  
    # ...  
    'cms.plugins.twitter',  
)
```

```
)  
    # ...
```

### 1.6.11 Inherit

Displays all plugins of an other page or an other language. Great if you need always the same plugins on a lot of pages.

For installation be sure you have the following in your project's `INSTALLED_APPS` setting:

```
INSTALLED_APPS = (  
    # ...  
    'cms.plugins.inherit',  
    # ...  
)
```

**Warning:** The inherit plugin is currently the only core-plugin which can **not** be used in non-cms placeholders.

## 2.1 Internationalization

### 2.1.1 Multilingual URL Middleware

The multilingual URL middleware adds a language prefix to every URL.

Example:

```
/de/account/login/
/fr/account/login/
```

It also adds this prefix automatically to every href and form tag. To install it, include `'cms.middleware.multilingual.MultilingualURLMiddleware'` in your project's `MIDDLEWARE_CLASSES` setting.

### 2.1.2 Language Chooser

The `language_chooser` template tag will display a language chooser for the current page. You can modify the template in `menu/language_chooser.html` or provide your own template if necessary.

Example:

```
{% load menu_tags %}
{% language_chooser "myapp/language_chooser.html" %}
```

If the current URL is not handled by the CMS and you have some i18n slugs in the URL you may use the `set_language_changer` function in the view that handles the current URL.

In the models of the current object add an optional language parameter to the `get_absolute_url` function:

```
from django.utils.translation import get_language

def get_absolute_url(self, language=None):
    if not language:
        language = get_language()
    return reverse("product_view", args=[self.get_slug(language=language)])
```

In the view pass the `get_absolute_url` function to the `set_language_changer` function:

```
from cms.utils import set_language_changer

def get_product(request, slug):
    item = get_object_or_404(Product, slug=slug, published=True)
    set_language_changer(request, item.get_absolute_url)
    # ...
```

This allows the language chooser to have another URL than the current one. If the current URL is not handled by the CMS and no `set_language_changer` function is provided it will take the exact same URL as the current one and will only change the language prefix.

For the language chooser to work the Multilingual URL Middleware must be enabled.

### 2.1.3 page\_language\_url

This `template_tag` returns the URL of the current page in another language.

Example:

```
{% page_language_url "de" %}
```

### 2.1.4 CMS\_HIDE\_UNTRANSLATED

If you put `CMS_HIDE_UNTRANSLATED = False` in your `settings.py` all pages will be displayed in all languages even if they are not translated yet.

If `CMS_HIDE_UNTRANSLATED = True` is in your `settings.py`. And you are on a page that hasn't got a english translation yet and you view the german version then the language chooser will redirect to /. The same goes for urls that are not handled by the cms and display a language chooser.

## 2.2 Sitemap Guide

### 2.2.1 Sitemap

Sitemaps are XML files used by Google to index your website by using their **Webmaster Tools** and telling them the location of your sitemap.

The `CMSSitemap` will create a sitemap with all the published pages of your cms

### 2.2.2 Configuration

Add `django.contrib.sitemaps` to your project's `INSTALLED_APPS` setting. Add `from cms.sitemaps import CMSSitemap` to the top of your main `urls.py`. Add `url(r'^sitemap.xml$', 'django.contrib.sitemaps.views.sitemap', {'sitemaps': {'cmspages': CMSSitemap}})` to your `urlpatterns`.

### 2.2.3 Django.contrib.sitemaps

More information about `django.contrib.sitemaps` can be found in the official [Django documentation](#).

## 2.3 Templatetags

To use any of the following templatetags you need to load them first at the top of your template:

```
{% load cms_tags menu_tags %}
```

### 2.3.1 placeholder

The `placeholder` templatetag defines a placeholder on a page. All placeholders in a template will be auto-detected and can be filled with plugins when editing a page that is using said template. When rendering, the content of these plugins will appear where the `placeholder` tag was.

Example:

```
{% placeholder "content" %}
```

If you want additional content to be displayed in case the placeholder is empty, use the `or` argument and an additional `{% endplaceholder %}` closing tag. Everything between `{% placeholder "..."` or `{% endplaceholder %}` is rendered instead if the placeholder has no plugins or the plugins do not generate any output.

Example:

```
{% placeholder "content" or %}There is no content.{% endplaceholder %}
```

If you want to add extra variables to the context of the placeholder, you should use Django's `with` tag. For instance, if you want to resize images from your templates according to a context variable called `width`, you can pass it as follows:

```
{% with 320 as width %}{% placeholder "content" %}{% endwith %}
```

If you want the placeholder to inherit the content of a placeholder with the same name on parent pages, simply pass the `inherit` argument:

```
{% placeholder "content" inherit %}
```

This will walk the page tree up till the root page and will show the first placeholder it can find with content.

It's also possible to combine this with the `or` argument to show an ultimate fallback if the placeholder and non of the placeholders on parent pages have plugins that generate content:

```
{% placeholder "content" inherit or %}There is no spoon.{% endplaceholder %}
```

See also the `PLACEHOLDER_CONF` setting where you can also add extra context variables and change some other placeholder behavior.

### 2.3.2 show\_placeholder

Displays a specific placeholder from a given page. This is useful if you want to have some more or less static content that is shared among many pages, such as a footer.

Arguments:

- `placeholder_name`
- `page_lookup` (see [Page Lookup](#) for more information)
- `language` (optional)
- `site` (optional)

Examples:

```
{% show_placeholder "footer" "footer_container_page" %}
{% show_placeholder "content" request.current_page.parent_id %}
{% show_placeholder "teaser" request.current_page.get_root %}
```

#### Page Lookup

The `page_lookup` argument, passed to several templatetags to retrieve a page, can be of any of the following types:

- String: interpreted as the `reverse_id` field of the desired page, which can be set in the “Advanced” section when editing a page.
- Integer: interpreted as the primary key (pk field) of the desired page
- dict: a dictionary containing keyword arguments to find the desired page (for instance: `{'pk': 1}`)
- Page: you can also pass a page object directly, in which case there will be no database lookup.

If you know the exact page you are referring to, it is a good idea to use a `reverse_id` (a string used to uniquely name a page) rather than a hard-coded numeric ID in your template. For example, you might have a help page that you want to link to or display parts of on all pages. To do this, you would first open the help page in the admin interface and enter an ID (such as `help`) under the ‘Advanced’ tab of the form. Then you could use that `reverse_id` with the appropriate templatetags:

```
{% show_placeholder "right-column" "help" %}
<a href="{% page_url "help" %}">Help page</a>
```

If you are referring to a page *relative* to the current page, you’ll probably have to use a numeric page ID or a page object. For instance, if you want the content of the parent page display on the current page, you can use:

```
{% show_placeholder "content" request.current_page.parent_id %}
```

Or, suppose you have a placeholder called `teaser` on a page that, unless a content editor has filled it with content specific to the current page, should inherit the content of its root-level ancestor:

```
{% placeholder "teaser" or %}
  {% show_placeholder "teaser" request.current_page.get_root %}
{% endplaceholder %}
```

### 2.3.3 show\_uncached\_placeholder

The same as `show_placeholder`, but the placeholder contents will not be cached.

Arguments:

- `placeholder_name`
- `page_lookup` (see *Page Lookup* for more information)
- `language` (optional)
- `site` (optional)

Example:

```
{% show_uncached_placeholder "footer" "footer_container_page" %}
```

### 2.3.4 plugins\_media

Outputs the appropriate tags to include all media that is used by the plugins on a page (defined using the `Media` class in the plugin class).

You normally want to place this in your `<head>` tag.

Example:

```
{% plugins_media %}
```

Arguments:

- `page_lookup` (optional; see *Page Lookup* for more information)

If you need to include the media from another page, for instance if you are using a placeholder from another page using the `show_placeholder` tag, you can supply the `page_lookup` attribute to indicate the page in question:



```
{% plugins_media "teaser" %}
```

For a reference on what plugin media is required by a specific plugin, look at that plugin's reference.

### 2.3.5 page\_url

Displays the URL of a page in the current language.

Arguments:

- `page_lookup` (see *Page Lookup* for more information)

Example:

```
<a href="{% page_url "help" %}">Help page</a>
<a href="{% page_url request.current_page.parent %}">Parent page</a>
```

### 2.3.6 page\_attribute

This templatetag is used to display an attribute of the current page in the current language.

Arguments:

- `attribute_name`
- `page_lookup` (optional; see *Page Lookup* for more information)

Possible values for `attribute_name` are: "title", "menu\_title", "page\_title", "slug", "meta\_description", "meta\_keywords" (note that you can also supply that argument without quotes, but this is deprecated because the argument might also be a template variable).

Example:

```
{% page_attribute "page_title" %}
```

If you supply the optional `page_lookup` argument, you will get the page attribute from the page found by that argument.

Example:

```
{% page_attribute "page_title" "my_page_reverse_id" %}
{% page_attribute "page_title" request.current_page.parent_id %}
{% page_attribute "slug" request.current_page.get_root %}
```

### 2.3.7 show\_menu

The `show_menu` tag renders the navigation of the current page. You can overwrite the appearance and the HTML if you add a `cms/menu.html` template to your project or edit the one provided with `django-cms`. `show_menu` takes four optional parameters: `start_level`, `end_level`, `extra_inactive`, and `extra_active`.

The first two parameters, `start_level` (default=0) and `end_level` (default=100) specify from what level to which level should the navigation be rendered. If you have a home as a root node and don't want to display home you can render the navigation only after level 1.

The third parameter, `extra_inactive` (default=0), specifies how many levels of navigation should be displayed if a node is not a direct ancestor or descendant of the current active node.

Finally, the fourth parameter, `extra_active` (default=100), specifies how many levels of descendants of the currently active node should be displayed.

## Some Examples

Complete navigation (as a nested list):

```
<ul>
  {% show_menu 0 100 100 100 %}
</ul>
```

Navigation with active tree (as a nested list):

```
<ul>
  {% show_menu 0 100 0 100 %}
</ul>
```

Navigation with only one active extra level:

```
<ul>
  {% show_menu 0 100 0 1 %}
</ul>
```

Level 1 navigation (as a nested list):

```
<ul>
  {% show_menu 1 %}
</ul>
```

Navigation with a custom template:

```
{% show_menu 0 100 100 100 "myapp/menu.html" %}
```

### 2.3.8 show\_menu\_below\_id

If you have set an id in the advanced settings of a page, you can display the submenu of this page with a template tag. For example, we have a page called meta that is not displayed in the navigation and that has the id “meta”:

```
<ul>
  {% show_menu_below_id "meta" %}
</ul>
```

You can give it the same optional parameters as show\_menu:

```
<ul>
  {% show_menu_below_id "meta" 0 100 100 100 "myapp/menu.html" %}
</ul>
```

### 2.3.9 show\_sub\_menu

Displays the sub menu of the current page (as a nested list). Takes one argument that specifies how many levels deep should the submenu be displayed. The template can be found at cms/sub\_menu.html:

```
<ul>
  {% show_sub_menu 1 %}
</ul>
```

Or with a custom template:

```
<ul>
  {% show_sub_menu 1 "myapp/submenu.html" %}
</ul>
```

### 2.3.10 show\_breadcrumb

Renders the breadcrumb navigation of the current page. The template for the HTML can be found at `cms/breadcrumb.html`:

```
{% show_breadcrumb %}
```

Or with a custom template and only display level 2 or higher:

```
{% show_breadcrumb 2 "myapp/breadcrumb.html" %}
```

Usually, only pages visible in the navigation are shown in the breadcrumb. To include *all* pages in the breadcrumb, write:

```
{% show_breadcrumb 0 "cms/breadcrumb.html" 0 %}
```

If the current URL is not handled by the CMS or by a navigation extender, the current menu node can not be determined. In this case you may need to provide your own breadcrumb via the template. This is mostly needed for pages like login, logout and third-party apps. This can easily be accomplished by a block you overwrite in your templates.

For example in your `base.html`:

```
<ul>
  {% block breadcrumb %}
  {% show_breadcrumb %}
  {% endblock %}
</ul>
```

And then in your app template:

```
{% block breadcrumb %}
<li><a href="/">home</a></li>
<li>My current page</li>
{% endblock %}
```

### 2.3.11 page\_language\_url

Returns the url of the current page in an other language:

```
{% page_language_url de %}
{% page_language_url fr %}
{% page_language_url en %}
```

If the current url has no cms-page and is handled by a navigation extender and the url changes based on the language: You will need to set a `language_changer` function with the `set_language_changer` function in `cms.utils`.

For more information, see [Internationalization](#).

### 2.3.12 language\_chooser

The `language_chooser` template tag will display a language chooser for the current page. You can modify the template in `menu/language_chooser.html` or provide your own template if necessary.

Example:

```
{% language_chooser %}
```

or with custom template:

```
{% language_chooser "myapp/language_chooser.html" %}
```

The `language_chooser` has three different modes in which it will display the languages you can choose from: “raw” (default), “native”, “current” and “short”. It can be passed as last argument to the `language_chooser` tag as a string. In “raw” mode, the language will be displayed like it’s verbose name in the settings. In “native” mode the languages are displayed in their actual language (eg. German will be displayed “Deutsch”, Japanese as “” etc). In “current” mode the languages are translated into the current language the user is seeing the site in (eg. if the site is displayed in German, Japanese will be displayed as “Japanisch”). “Short” mode takes the language code (eg. “en”) to display.

If the current url has no cms-page and is handled by a navigation extender and the url changes based on the language: You will need to set a `language_changer` function with the `set_language_changer` function in `cms.utils`.

For more information, see [Internationalization](#).

---

## Extending the CMS

---

### 3.1 Extending the CMS: Examples

From this part onwards, this tutorial assumes you have done the [Django Tutorial](#) and we will show you how to integrate that poll app into the django CMS. If a poll app is mentioned here, we mean the one you get when finishing the [Django Tutorial](#).

We assume your main `urls.py` looks somewhat like this:

```
from django.conf.urls.defaults import *

from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    (r'^admin/', include(admin.site.urls)),
    (r'^polls/', include('polls.urls')),
    (r'^$', include('cms.urls')),
)
```

#### 3.1.1 My First Plugin

A Plugin is a small bit of content you can place on your pages.

##### The Model

For our polling app we would like to have a small poll plugin, that shows one poll and let's the user vote.

In your poll application's `models.py` add the following model:

```
from cms.models import CMSPlugin

class PollPlugin(CMSPlugin):
    poll = models.ForeignKey('polls.Poll', related_name='plugins')

    def __unicode__(self):
        return self.poll.question
```

---

**Note:** django CMS Plugins must inherit from `cms.models.CMSPlugin` (or a subclass thereof) and not `django.db.models.Model`.

---

Run `syncdb` to create the database tables for this model or see [Using South with Django-CMS](#) to see how to do it using [South](#)

### The Plugin Class

Now create a file `cms_plugins.py` in the same folder your `models.py` is in, so following the Django Tutorial, your polls app folder should look like this now:

```
polls/
  __init__.py
  cms_plugins.py
  models.py
  tests.py
  views.py
```

The plugin class is responsible to provide the django CMS with the necessary information to render your Plugin.

For our poll plugin, write following plugin class:

```
from cms.plugin_base import CMSPluginBase
from cms.plugin_pool import plugin_pool
from polls.models import PollPlugin as PollPluginModel
from django.utils.translation import ugettext as _

class PollPlugin(CMSPluginBase):
    model = PollPluginModel # Model where data about this plugin is saved
    name = _("Poll Plugin") # Name of the plugin
    render_template = "polls/plugin.html" # template to render the plugin with

    def render(self, context, instance, placeholder):
        context.update({'instance':instance})
        return context

plugin_pool.register_plugin(PollPlugin) # register the plugin
```

---

**Note:** All plugin classes must inherit from `cms.plugin_base.CMSPluginBase` and must register themselves with the `cms.plugin_pool.plugin_pool`.

---

### The Template

You probably noticed the `render_template` attribute on that plugin class, for our plugin to work, that template must exist and is responsible for rendering the plugin.

The template could look like this:

```
<h1>{{ poll.question }}</h1>

<form action="{% url polls.views.vote poll.id %}" method="post">
  {% csrf_token %}
  {% for choice in poll.choice_set.all %}
    <input type="radio" name="choice" id="choice{{ forloop.counter }}" value="{{ choice.id }}" />
    <label for="choice{{ forloop.counter }}">{{ choice.choice }}</label><br />
  {% endfor %}
  <input type="submit" value="Vote" />
</form>
```

---

**Note:** We don't show the errors here, because when submitting the form you're taken off this page to the actual voting page.

---

### 3.1.2 My First App (apphook)

Right now, external apps are statically hooked into the main `urls.py`, that is not the preferred way in the django CMS. Ideally you attach your apps to CMS Pages.

For that purpose you write CMS Apps. That is just a small class telling the CMS how to include that app.

CMS Apps live in a file called `cms_app.py`, so go ahead and create that to make your polls app look like this:

```
polls/
  __init__.py
  cms_app.py
  cms_plugins.py
  models.py
  tests.py
  views.py
```

In this file, write:

```
from cms.app_base import CMSApp
from cms.apphook_pool import apphook_pool
from django.utils.translation import ugettext_lazy as _

class PollsApp(CMSApp):
    name = _("Poll App") # give your app a name, this is required
    urls = ["polls.urls"] # link your app to url configuration(s)

apphook_pool.register(PollsApp) # register your app
```

Now remove the inclusion of the polls urls in your main `urls.py` so it looks like this:

```
from django.conf.urls.defaults import *

from django.contrib import admin
admin.autodiscover()

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

Now open your admin in your browser and edit a CMS Page. Open the ‘Advanced Settings’ tab and choose ‘Polls App’ for your ‘Application’.

Django administration Welcome, admin. Change password / Log out

Home > Cms > Pages > Polls

**Change page** (you can perform actions on this page directly) History View on site →

English Japanese

**Title:**  Menu Title:   
The default title Overwrite what is displayed in the menu

**Slug:**   
The part of the title that is used in the URL

Is published  In navigation

Publication date: **Date:**  Today |   
**Time:**  Now |   
When the page should go live. Status must be "Published" for page to go live.

Publication end date: **Date:**  Today |   
**Time:**  Now |   
When to expire the page. Leave empty to never expire.

**Basic Settings**  
 Note: This page reloads if you change the selection. Save it first.  
**Template:**   
The template used to render the content.

**Col Content**  
  No Plugins present. Add a plugin to this placeholder-slot.

**Feature**  
  No Plugins present. Add a plugin to this placeholder-slot.

**Advanced Settings (Hide)**

**Id:**   
An unique identifier that is used with the page\_url templatetag for linking to this page

**Overwrite URL:**   
Keep this field empty if standard path should be used.

**Redirect:**   
Redirects to this URL.

Login required

**Menu visibility:**   
limit when this page is visible in the menu

Soft root  
All ancestors will not be displayed in the navigation

**Application:**   to this page.

**SEO Settings (Show)**

Now for those changes to take effect, unfortunately you will have to restart your server. So do that and now if you navigate to that CMS Page, you will see your polls application.

### 3.1.3 My First Menu

Now you might have noticed that the menu tree stops at the CMS Page you created in the last step, so let's create a menu that shows a node for each poll you have active.

For this we need a file called `menu.py`, create it and check your polls app looks like this:

```
polls/
  __init__.py
  cms_app.py
  cms_plugins.py
  menu.py
  models.py
  tests.py
  views.py
```

In your `menu.py` write:

```
from cms.menu_bases import CMSAttachMenu
from menus.base import Menu, NavigationNode
from menus.menu_pool import menu_pool
from django.core.urlresolvers import reverse
from django.utils.translation import ugettext_lazy as _
```



```

from polls.models import Poll

class PollsMenu(CMSAttachMenu):
    name = _("Polls Menu") # give the menu a name, this is required.

    def get_nodes(self, request):
        """
        This method is used to build the menu tree.
        """
        nodes = []
        for poll in Poll.objects.all():
            # the menu tree consists of NavigationNode instances
            # Each NavigationNode takes a label as first argument, a URL as
            # second argument and a (for this tree) unique id as third
            # argument.
            node = NavigationNode(
                poll.question,
                reverse('polls.views.detail', args=(poll.pk,)),
                poll.pk
            )
            nodes.append(node)
        return nodes
menu_pool.register_menu(PollsMenu) # register the menu.

```

Now this menu alone doesn't do a whole lot yet, we have to attach it to the Apphook first.

So open your `cms_apps.py` and write:

```

from cms.app_base import CMSApp
from cms.apphook_pool import apphook_pool
from polls.menu import PollsMenu
from django.utils.translation import ugettext_lazy as _

class PollsApp(CMSApp):
    name = _("Poll App")
    urls = ["polls.urls"]
    menu = [PollsMenu] # attach a CMSAttachMenu to this apphook.

apphook_pool.register(PollsApp)

```

## 3.2 Custom Plugins

You have three options to extend Django CMS: Custom plugins, plugin context processors, and plugin processors.

### 3.2.1 Writing a custom plugin

You can use `python manage.py startapp` to get some basefiles for your plugin, or just add a folder `gallery` to your project's root folder, add an empty `__init__.py`, so that the module gets detected.

Suppose you have the following gallery model:

```

class Gallery(models.Model):
    name = models.CharField(max_length=30)

class Picture(models.Model):
    gallery = models.ForeignKey(Gallery)
    image = models.ImageField(upload_to="uploads/images/")
    description = models.CharField(max_length=60)

```

And that you want to display this gallery between two text blocks.

You can do this with a CMS plugin. To create a CMS plugin you need two components: a CMSPlugin model and a cms\_plugins.py file.

### Plugin Model

First create a model that links to the gallery via a ForeignKey field:

```
from cms.models import CMSPlugin

class GalleryPlugin(CMSPlugin):
    gallery = models.ForeignKey(Gallery)
```

Be sure that your model inherits the CMSPlugin class. The plugin model can have any fields it wants. They are the fields that get displayed if you edit the plugin.

Now models.py looks like the following:

```
from django.db import models
from cms.models import CMSPlugin

class Gallery(models.Model):
    parent = models.ForeignKey('self', blank=True, null=True)
    name = models.CharField(max_length=30)

    def __unicode__(self):
        return self.name

    def get_absolute_url(self):
        return reverse('gallery_view', args=[self.pk])

    class Meta:
        verbose_name_plural = 'gallery'

class Picture(models.Model):
    gallery = models.ForeignKey(Gallery)
    image = models.ImageField(upload_to="uploads/images/")
    description = models.CharField(max_length=60)

class GalleryPlugin(CMSPlugin):
    gallery = models.ForeignKey(Gallery)
```

**Warning:** CMSPlugin subclasses cannot be further subclassed, if you want to make a reusable plugin model, make an abstract base model which does not extend CMSPlugin and subclass this abstract model as well as CMSPlugin in your real plugin model. Further note that you cannot name your model fields the same as any plugin's lowercased model name you use is called, due to the implicit one to one relation Django uses for subclassed models.

### Handling Relations

If your custom plugin has foreign key or many-to-many relations you are responsible for copying those if necessary whenever the CMS copies the plugin.

To do this you can implement a method called `copy_relations` on your plugin model which gets the *old* instance of the plugin as argument.

Lets assume this is your plugin:

```
class ArticlePluginModel(CMSPlugin):
    title = models.CharField(max_length=50)
    sections = models.ManyToManyField(Section)

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

Now when the plugin gets copied, you want to make sure the sections stay:

```
def copy_relations(self, oldinstance):
    self.sections = oldinstance.sections.all()
```

Your full model now:

```
class ArticlePluginModel(CMSPlugin):
    title = models.CharField(max_length=50)
    sections = models.ManyToManyField(Section)

    def __unicode__(self):
        return self.title

    def copy_relations(self, oldinstance):
        self.sections = oldinstance.sections.all()
```

## cms\_plugins.py

After that create in the application folder (the same one where models.py is) a cms\_plugins.py file.

In there write the following:

```
from cms.plugin_base import CMSPluginBase
from cms.plugin_pool import plugin_pool
from models import GalleryPlugin
from django.utils.translation import ugettext as _

class CMSGalleryPlugin(CMSPluginBase):
    model = GalleryPlugin
    name = _("Gallery")
    render_template = "gallery/gallery.html"

    def render(self, context, instance, placeholder):
        context.update({
            'gallery':instance.gallery,
            'object':instance,
            'placeholder':placeholder
        })
        return context

plugin_pool.register_plugin(CMSGalleryPlugin)
```

CMSPluginBase itself inherits from ModelAdmin so you can use all the things (inlines for example) you would use in a regular admin class.

For a list of all the options you have on CMSPluginBase have a look at the plugin reference

## Template

Now create a gallery.html template in templates/gallery/ and write the following in there:

```
{% for image in gallery.picture_set.all %}
    
{% endfor %}
```

Add a file `admin.py` in your plugin root-folder and insert the following:

```
from django.contrib import admin
from cms.admin.placeholderadmin import PlaceholderAdmin
from models import Gallery, Picture

class PictureInline(admin.StackedInline):
    model = Picture

class GalleryAdmin(admin.ModelAdmin):
    inlines = [PictureInline]

admin.site.register(Gallery, GalleryAdmin)
```

Now go into the admin create a gallery and afterwards go into a page and add a gallery plugin and some pictures should appear in your page.

### Limiting Plugins per Placeholder

You can limit in which placeholder certain plugins can appear. Add a `CMS_PLACEHOLDER_CONF` to your `settings.py`.

Example:

```
CMS_PLACEHOLDER_CONF = {
    'col_sidebar': {
        'plugins': ('FilePlugin', 'FlashPlugin', 'LinkPlugin', 'PicturePlugin', 'TextPlugin', 'Sn
        'name': gettext("sidebar column")
    },
    'col_left': {
        'plugins': ('FilePlugin', 'FlashPlugin', 'LinkPlugin', 'PicturePlugin', 'TextPlugin', 'Sn
        'name': gettext("left column")
    },
    'col_right': {
        'plugins': ('FilePlugin', 'FlashPlugin', 'LinkPlugin', 'PicturePlugin', 'TextPlugin', 'Sn
        'name': gettext("right column")
    },
}
```

“`col_left`” and “`col_right`” are the names of two placeholders. The plugins list are filled with Plugin class names you find in the `cms_plugins.py`. You can add extra context to each placeholder so plugin-templates can react to them.

You can change the displayed name in the admin with the `name` parameter. In combination with `gettext` you can translate this names according to the language of the user. Additionally you can limit the number of plugins (either total or by type) for each placeholder with the `limits` parameter (see [Configuration](#) for details).

### Advanced

`CMSSGalleryPlugin` can be even further customized:

Because `CMSSPluginBase` extends `ModelAdmin` from `django.contrib.admin` you can use all the things you are used to with normal admin classes. You can define inlines, the form, the form template etc.

Note: If you want to overwrite the form be sure to extend from `admin/cms/page/plugin_change_form.html` to have an unified look across the plugins and to have the preview functionality automatically installed.

### 3.2.2 Plugin Context Processors

Plugin context processors are callables that modify all plugin's context before rendering. They are enabled using the `CMS_PLUGIN_CONTEXT_PROCESSORS` setting.

A plugin context processor takes 2 arguments:

**instance:**

The instance of the plugin model

**placeholder:**

The instance of the placeholder this plugin appears in.

The return value should be a dictionary containing any variables to be added to the context.

Example:

```
# settings.py:
CMS_PLUGIN_CONTEXT_PROCESSORS = (
    'yourapp.cms_plugin_context_processors.add_verbose_name',
)

# yourapp.cms_plugin_context_processors.py:
def add_verbose_name(instance, placeholder):
    """
    This plugin context processor adds the plugin model's verbose_name to context.
    """
    return {'verbose_name': instance._meta.verbose_name}
```

### 3.2.3 Plugin Processors

Plugin processors are callables that modify all plugin's output after rendering. They are enabled using the `CMS_PLUGIN_PROCESSORS` setting.

A plugin processor takes 4 arguments:

**instance:**

The instance of the plugin model

**placeholder:**

The instance of the placeholder this plugin appears in.

**rendered\_content:**

A string containing the rendered content of the plugin.

**original\_context:**

The original context for the template used to render the plugin.

Note that plugin processors are also applied to plugins embedded in Text. Depending on what your processor does, this might break the output. For example, if your processor wraps the output in a DIV tag, you might end up having DIVs inside of P tags, which is invalid. You can prevent such cases by returning `rendered_content` unchanged if `instance._render_meta.text_enabled` is True, which is the case when rendering an embedded plugin.

#### Example

Suppose you want to put wrap each plugin in the main placeholder in a colored box, but it would be too complicated to edit each individual plugin's template:

In your settings.py:

```
CMS_PLUGIN_PROCESSORS = (
    'yourapp.cms_plugin_processors.wrap_in_colored_box',
)
```

In your `yourapp.cms_plugin_processors.py`:

```
def wrap_in_colored_box(instance, placeholder, rendered_content, original_context):
    '''
    This plugin processor wraps each plugin's output in a colored box if it is in the "main" placeholder.
    '''
    if placeholder.slot != 'main' \
        or (instance._render_meta.text_enabled and instance.parent):
        # Plugins not in the main placeholder should remain unchanged
        # Plugins embedded in Text should remain unchanged
        return rendered_content
    else:
        from django.template import Context, Template
        # For simplicity's sake, construct the template from a string:
        t = Template('<div style="border: 10px {{ border_color }} solid; background-color: {{ background_color }};">{{ content }}</div>')
        # Prepare that template's context:
        c = Context({
            'content': rendered_content,
            # Some plugin models might allow you to customize the colors,
            # for others, use default colors:
            'background_color': instance.background_color if hasattr(instance, 'background_color') else 'lightgray',
            'border_color': instance.border_color if hasattr(instance, 'border_color') else 'lightgray',
        })
        # Finally, render the content through that template, and return the output
        return t.render(c)
```

## 3.3 App Integration

It is pretty easy to integrate your own Django applications with `django-cms`. You have 5 ways of integrating your app:

1. Menus
  - Static extend the menu entries
2. AttachMenus
  - Attach your menu to a page.
3. App-Hooks
  - Attach whole apps with optional menu to a page.
4. Navigation Modifiers
  - Modify the whole menu tree
5. Custom Plugins
  - Display your models / content in cms pages

### 3.3.1 Menus

Create a `menu.py` in your application and write the following inside:

```
from menus.base import Menu, NavigationNode
from menus.menu_pool import menu_pool
from django.utils.translation import gettext_lazy as _

class TestMenu(Menu):
```

```

def get_nodes(self, request):
    nodes = []
    n = NavigationNode(_('sample root page'), "/", 1)
    n2 = NavigationNode(_('sample settings page'), "/bye/", 2)
    n3 = NavigationNode(_('sample account page'), "/hello/", 3)
    n4 = NavigationNode(_('sample my profile page'), "/hello/world/", 4, 3)
    nodes.append(n)
    nodes.append(n2)
    nodes.append(n3)
    nodes.append(n4)
    return nodes

```

```
menu_pool.register_menu(TestMenu)
```

If you refresh a page you should now see the menu entries from above. The `get_nodes` function should return a list of `NavigationNode` instances. A `NavigationNode` takes the following arguments:

- title
  - What should the menu entry read?
- url,
  - Link if menu entry is clicked.
- id
  - A unique id for this menu.
- parent\_id=None
  - If this is a child of an other node give here the id of the parent.
- parent\_namespace=None
  - If the parent node is not from this menu you can give it the parent namespace. The namespace is the name of the class. In the above example that would be: "TestMenu"
    - attr=None
  - A dictionary of additional attributes you may want to use in a modifier or in the template.

### 3.3.2 Attach Menus

Classes that extend from `Menu` always get attached to the root. But if you want the menu be attached to a CMS-page you can do that as well.

Instead of extending from `Menu` you need to extend from `CMSAttachMenu` and you need to define a name. We will do that with the example from above:

```

from menus.base import NavigationNode
from menus.menu_pool import menu_pool
from django.utils.translation import ugettext_lazy as _
from cms.menu_bases import CMSAttachMenu

class TestMenu(CMSAttachMenu):

    name = _("test menu")

    def get_nodes(self, request):
        nodes = []
        n = NavigationNode(_('sample root page'), "/", 1)
        n2 = NavigationNode(_('sample settings page'), "/bye/", 2)
        n3 = NavigationNode(_('sample account page'), "/hello/", 3)
        n4 = NavigationNode(_('sample my profile page'), "/hello/world/", 4, 3)
        nodes.append(n)

```

```

nodes.append(n2)
nodes.append(n3)
nodes.append(n4)
return nodes

menu_pool.register_menu(TestMenu)

```

Now you can link this Menu to a page in the ‘Advanced’ tab of the page settings under attached menu.

It is encouraged to use [django-mptt](#) (a suitable version is included in the *mptt* directory) for the tree structure because of performance considerations. The objects provided must adhere to the following structure:

Each must have a `get_menu_title` function, a `get_absolute_url` function, and a `childrens` array with all of its children inside (the ‘s’ at the end of `childrens` is done on purpose because `children` is already taken by `mptt`).

Be sure that `get_menu_title` and `get_absolute_url` don’t trigger any queries when called in a template or you may have some serious performance and DB problems with a lot of queries.

It may be wise to cache the output of `get_nodes`. For this you may need to write a wrapper class because of dynamic content that the pickle module can’t handle.

If you want to display some static pages in the navigation (“login”, for example) you can write your own “dummy” class that adheres to the conventions described above.

A base class for this purpose can be found in `cms/utils/navigation.py`

### 3.3.3 App-Hooks

With App-Hooks you can attach whole Django applications to pages. For example you have a news app and you want it attached to your news page.

To create an apphook create a `cms_app.py` in your application. And in there write the following:

```

from cms.app_base import CMSApp
from cms.apphook_pool import apphook_pool
from django.utils.translation import ugettext_lazy as _

class MyApphook(CMSApp):
    name = _("My Apphook")
    urls = ["myapp.urls"]

apphook_pool.register(MyApphook)

```

Replace “myapp.urls” with the path to your applications `urls.py`.

Now edit a page and open the advanced settings tab. Select your new apphook under “Application”. Save the page.

**\*\* ATTENTION \*\*** If you are on a multi-threaded server (mostly all webservers, except the dev-server): Restart the server because the URLs are cached by Django and in a multi-threaded environment we don’t know which caches are cleared yet.

If you attached the app to a page with the url `/hello/world/` and the app has a `urls.py` that looks like this:

```

from django.conf.urls.defaults import *

urlpatterns = patterns('sampleapp.views',
    url(r'^$', 'main_view', name='app_main'),
    url(r'^sublevel/$', 'sample_view', name='app_sublevel'),
)

```

The ‘main\_view’ should now be available at `/hello/world/` and the ‘sample\_view’ has the url `/hello/world/sublevel/`.



---

**Note:** All views that are attached like this must return a RequestContext instance instead of the default Context instance.

---

### Language Namespaces

An additional feature of apphooks is that if you use the MultilingualURLMiddleware all apphook urls are language namespaced.

What this means:

To reverse the first url from above you would use something like this in your template:

```
{% url app_main %}
```

If you want to access the same url but in a different language use a language namespace:

```
{% url de:app_main %}
{% url en:app_main %}
{% url fr:app_main %}
```

If you want to add a menu to that page as well that may represent some views in your app add it to your apphook like this:

```
from myapp.menu import MyAppMenu

class MyApphook (CMSApp) :
    name = _("My Apphook")
    urls = ["myapp.urls"]
    menus = [MyAppMenu]

apphook_pool.register(MyApphook)
```

For an example if your app has a Category model and you want this category model to be displayed in the menu when you attach the app to a page. We assume the following model:

```
from django.db import models
from django.core.urlresolvers import reverse
import mptt

class Category(models.Model):
    parent = models.ForeignKey('self', blank=True, null=True)
    name = models.CharField(max_length=20)

    def __unicode__(self):
        return self.name

    def get_absolute_url(self):
        return reverse('category_view', args=[self.pk])

try:
    mptt.register(Category)
except mptt.AlreadyRegistered:
    pass
```

It is encouraged to use `django-mptt` (a suitable version is included in the `mptt` directory) if you have data that is organized in a tree.

We would now create a menu out of these categories:

```
from menus.base import NavigationNode
from menus.menu_pool import menu_pool
from django.utils.translation import ugettext_lazy as _
from cms.menu_bases import CMSAttachMenu
from myapp.models import Category
```

```
class CategoryMenu(CMSAttachMenu):

    name = _("test menu")

    def get_nodes(self, request):
        nodes = []
        for category in Category.objects.all().order_by("tree_id", "lft"):
            nodes.append(NavigationNode(category.name, category.pk, category.parent_id))
        return nodes

menu_pool.register_menu(CategoryMenu)
```

If you add this menu now to your app-hook:

```
from myapp.menus import CategoryMenu

class MyApphook(CMSApp):
    name = _("My Apphook")
    urls = ["myapp.urls"]
    menus = [MyAppMenu, CategoryMenu]
```

You get the static entries of MyAppMenu and the dynamic entries of CategoryMenu both attached to the same page.

### 3.3.4 Navigation Modifiers

Navigation Modifiers can add or change properties of NavigationNodes, they even can rearrange whole menus. You normally want to create them in your apps menu.py.

A simple modifier looks something like this:

```
from menus.base import Modifier
from menus.menu_pool import menu_pool

class MyMode(Modifier):
    """
    """

    def modify(self, request, nodes, namespace, root_id, post_cut, breadcrumb):
        if post_cut:
            return nodes
        count = 0
        for node in nodes:
            node.counter = count
            count += 1
        return nodes
```

It has a function modify that should return a list of NavigationNodes. Modify should take the following arguments:

- request
  - A Django request instance. Maybe you want to modify based on sessions, or user or permissions?
- nodes
  - All the nodes. Normally you want to return them again.
- namespace
  - A Menu Namespace. Only given if somebody requested a menu with only nodes from this namespace.
- root\_id
  - Was a menu request based on an ID?

- `post_cut`

Every modifier is called 2 times. First on the whole tree. After that the tree gets cut. To only show the nodes that are shown in the current menu. After the cut the modifiers are called again with the final tree. If this is the case `post_cut` is `True`.

- `breadcrumb`

Is this not a menu call but a breadcrumb call?

Here is an example of a build in modifier that marks all nodes level:

```
class Level(Modifier):
    """
    marks all node levels
    """
    post_cut = True

    def modify(self, request, nodes, namespace, root_id, post_cut, breadcrumb):
        if breadcrumb:
            return nodes
        for node in nodes:
            if not node.parent:
                if post_cut:
                    node.menu_level = 0
                else:
                    node.level = 0
                self.mark_levels(node, post_cut)
        return nodes

    def mark_levels(self, node, post_cut):
        for child in node.children:
            if post_cut:
                child.menu_level = node.menu_level + 1
            else:
                child.level = node.level + 1
            self.mark_levels(child, post_cut)
```

### 3.3.5 Custom Plugins

If you want to display content of your apps on other pages custom plugins are a great way to accomplish that. For example, if you have a news app and you want to display the top 10 news entries on your homepage, a custom plugin is the way to go.

For a detailed explanation on how to write custom plugins please head over to the [Custom Plugins](#) section.

## 3.4 API References

### 3.4.1 cms.plugin\_base

```
class cms.plugin_base.CMSPluginBase
    Inherits django.contrib.admin.options.ModelAdmin.

    admin_preview
        Defaults to True, if False no preview is done in the admin.

    change_form_template
        Custom template to use to render the form to edit this plugin.

    form
        Custom form class to be used to edit this plugin.
```

**model**

Is the CMSPlugin model we created earlier. If you don't need a model because you just want to display some template logic, use CMSPlugin from `cms.models` as the model instead.

**module**

Will be group the plugin in the plugin editor. If module is None, plugin is grouped "Generic" group.

**name**

Will be displayed in the plugin editor.

**render\_plugin**

If set to `False`, this plugin will not be rendered at all.

**render\_template**

Will be rendered with the context returned by the render function.

**text\_enabled**

Whether this plugin can be used in text plugins or not.

**icon\_alt** (*instance*)

Returns the alt text for the icon used in text plugins, see `icon_src()`.

**icon\_src** (*instance*)

Returns the url to the icon to be used for the given instance when that instance is used inside a text plugin.

**render** (*context, instance, placeholder*)

This method returns the context to be used to render the template specified in `render_template`.

**Parameters**

- **context** – Current template context.
- **instance** – Plugin instance that is being rendered.
- **placeholder** – Name of the placeholder the plugin is in.

**Return type** dict

**class PluginMedia**

Defines media which is required to render this plugin.

**css**

The CSS files required to render this plugin as a dictionary with the display type as keys and a sequence of strings as values.

**js**

The Javascript files required to render this plugin as a sequence of strings.

### 3.4.2 menus.base

**class** `menus.base.NavigationNode` (*title, url, id[, parent\_id=None][, parent\_namespace=None][, attr=None][, visible=True]*)

A navigation node in a menu tree.

**Parameters**

- **title** (*string*) – The title to display this menu item with.
- **url** (*string*) – The URL associated with this menu item.
- **id** – Unique (for the current tree) ID of this item.
- **parent\_id** – Optional, ID of the parent item.
- **parent\_namespace** – Optional, namespace of the parent.
- **attr** (*dict*) – Optional, dictionary of additional information to store on this node.

- **visible** (*bool*) – Optional, defaults to `True`, whether this item is visible or not.

## 3.5 Placeholders outside the CMS

Placeholders are special model fields that DjangoCMS uses to render user-editable content (plugins) in templates. That is, it's the place where a user can add text, video or any other plugin to a webpage, using either the normal Django admin interface or the so called *frontend editing*.

Placeholders can be viewed as containers of `CMSPlugins`, and can be used outside the CMS in custom applications using the `PlaceholderField`.

By defining one (or several) `PlaceholderField` on a custom model you can take advantage of the full power of `CMSPlugins`, including frontend editing.

### 3.5.1 Quickstart

You need to define a `PlaceholderField` on the model you would like to use:

```
from django.db import models
from cms.models.fields import PlaceholderField

class MyModel(models.Model):
    # your fields
    my_placeholder = PlaceholderField('placeholder_name')
    # your methods
```

The `PlaceholderField` takes a string as first argument which will be used to configure which plugins can be used in this placeholder. The configuration is the same as for placeholders in the CMS.

If you install this model in the admin application, you have to use `PlaceholderAdmin` instead of `ModelAdmin` so the interface renders correctly:

```
from django.contrib import admin
from cms.admin.placeholderadmin import PlaceholderAdmin
from myapp import MyModel

admin.site.register(MyModel, PlaceholderAdmin)
```

Now to render the placeholder in a template you use the `render_placeholder` tag from the `placeholder_tags` template tag library:

```
{% load placeholder_tags %}

{% render_placeholder mymodel_instance.my_placeholder "640" %}
```

The `render_placeholder` tag takes a `PlaceholderField` instance as first argument and optionally accepts a width parameter as second argument for context sensitive plugins.

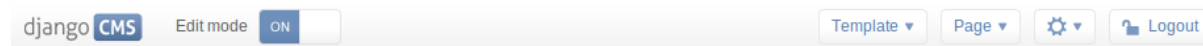
### 3.5.2 Adding content to a placeholder

There are two ways to add or edit content to a placeholder, the front-end admin view and the back-end view.

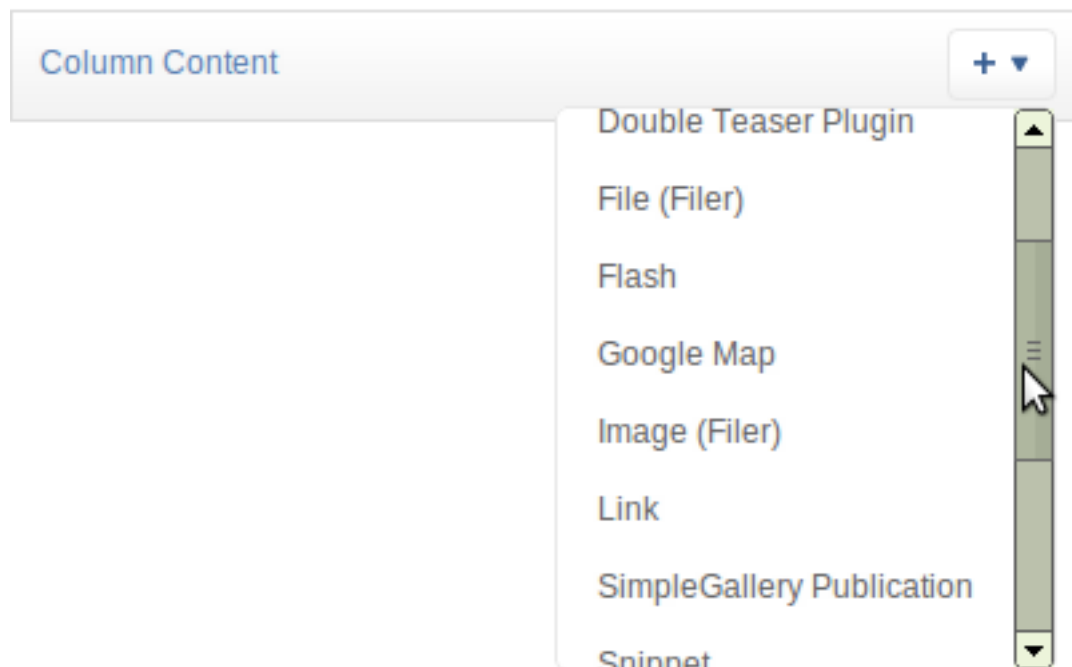
#### Using the front-end editor

Probably the most simple way to add content to a placeholder, simply visit the page displaying your model (where you put the `render_placeholder` tag), then append `?edit` to the page's URL. This will make a top banner appear, and after switching the "Edit mode" button to "on", the banner will prompt you for your username/password (the user should be allowed to edit the page, obviously)

You are now using the so-called *front-end edit mode*:



Once in Front-end editing mode, your placeholders should display a menu, allowing you to add plugins to them: the following screenshot shows a default selection of plugins in an empty placeholder.



Plugins are rendered at once, so you can have an idea what it will look like *in fine*, but to view the final look of a plugin simply leave edit mode by clicking the “Edit mode” button in the banner again.

### 3.5.3 Fieldsets

There are some hard restrictions if you want to add custom fieldsets to an admin page with at least one `PlaceholderField`:

1. Every `PlaceholderField` **must** be in it’s own fieldsets, one `PlaceholderField` per fieldset.
2. You **must** include the following two classes: `'plugin-holder'` and `'plugin-holder-nopage'`

## 3.6 Search and the Django-CMS

Currently the best way to integrate search with the Django-CMS is [Haystack](#), however it is not officially supported.

### 3.6.1 Haystack

If you go the Haystack way, you’ll need a `search_indexes.py`. Haystack doesn’t care if it’s in the same app as the models, so you can put it into any app within your project.

Here is an example **untested** and **unsupported** `search_indexes.py`:

```
from django.conf import settings
from django.utils.translation import string_concat, ugettext_lazy

from haystack import indexes, site
```

```

from cms.models.managers import PageManager
from cms.models.pagemodel import Page

def page_index_factory(lang, lang_name):
    if isinstance(lang_name, basestring):
        lang_name = ugettext_lazy(lang_name)

    def get_absolute_url(self):
        return '/%s%s' % (lang, Page.get_absolute_url(self))

    class Meta:
        proxy = True
        app_label = 'cms'
        verbose_name = string_concat(Page._meta.verbose_name, ' (' , lang_name, ')')
        verbose_name_plural = string_concat(Page._meta.verbose_name_plural, ' (' , lang_name, ')')

    attrs = {'__module__': Page.__module__,
            'Meta': Meta,
            'objects': PageManager(),
            'get_absolute_url': get_absolute_url}

    _PageProxy = type("Page%s" % lang.title() , (Page,), attrs)

    _PageProxy._meta.parent_attr = 'parent'
    _PageProxy._meta.left_attr = 'lft'
    _PageProxy._meta.right_attr = 'right'
    _PageProxy._meta.tree_id_attr = 'tree_id'

    class _PageIndex(indexes.SearchIndex):
        language = lang

        text = indexes.CharField(document=True, use_template=False)
        pub_date = indexes.DateTimeField(model_attr='publication_date')
        login_required = indexes.BooleanField(model_attr='login_required')
        url = indexes.CharField(stored=True, indexed=False, model_attr='get_absolute_url')
        title = indexes.CharField(stored=True, indexed=False, model_attr='get_title')

        def prepare(self, obj):
            self.prepared_data = super(_PageIndex, self).prepare(obj)
            plugins = obj.cmsplugin_set.filter(language=lang)
            text = ''
            for plugin in plugins:
                instance, _ = plugin.get_plugin_instance()
                if hasattr(instance, 'search_fields'):
                    text += ''.join(getattr(instance, field) for field in instance.search_fields)
            self.prepared_data['text'] = text
            return self.prepared_data

        def get_queryset(self):
            return _PageProxy.objects.published().filter(title_set__language=lang, publisher_is_d

    return _PageProxy, _PageIndex

for lang_tuple in settings.LANGUAGES:
    lang, lang_name = lang_tuple
    site.register(*page_index_factory(lang, lang_name))

```





---

## Contributing to the CMS

---

### 4.1 Contributing to Django-CMS

Like every open-source project, Django-CMS is always looking for motivated individuals to contribute to its source code. However, to ensure the highest code quality and keep the repository nice and tidy, everybody has to follow a few rules (nothing major, I promise :)

#### 4.1.1 Community

People interested in developing for the django-cms should join the [django-cms-developers](#) mailing list as well as heading over to #django-cms on freenode for help and to discuss the development.

You may also be interested in following @djangocmsstatus on twitter to get the github commits as well as the hudson build reports. There is also a @djangocms account for less technical announcements.

#### 4.1.2 In a nutshell

Here's what the contribution process looks like, in a bullet-points fashion, and only for the stuff we host on github:

1. django-cms is hosted on [Github](#), at <https://github.com/divio/django-cms>
2. The best method to contribute back is to create an account there, then fork the project. You can use this fork as if it was your own project, and should push your changes to it.
3. When you feel your code is good enough for inclusion, "send us a pull request", by using the nice Github web interface.

#### 4.1.3 Contributing Code

##### Getting the source code

If you're interested in developing a new feature for the cms, it is recommended that you first discuss it on the [django-cms-developers](#) mailing list so as not to do any work that will not get merged in anyway.

- Code will be reviewed and tested by at least one core developer, preferably by several. Other community members are welcome to give feedback.
- Code *must* be tested. Your pull request should include unit-tests (that cover the piece of code you're submitting, obviously)
- Documentation should reflect your changes if relevant. There is nothing worse than invalid documentation.
- Usually, if unit tests are written, pass, and your change is relevant, then it'll be merged.

Since we're hosted on github, django-cms uses git as a version control system.

The [Github help](#) is very well written and will get you started on using git and github in a jiffy. It is an invaluable resource for newbies and old timers alike.

### Syntax and conventions

We try to conform to [PEP8](#) as much as possible. A few highlights:

- Indentation should be exactly 4 spaces. Not 2, not 6, not 8. **4**. Also, tabs are evil.
- We try (loosely) to keep the line length at 79 characters. Generally the rule is "it should look good in a terminal-base editor" (eg vim), but we try not to be [Godwin's law] about it.

### Process

This is how you fix a bug or add a feature:

1. [fork](#) us on github.
2. Checkout your fork.
3. Hack hack hack, test test test, commit commit commit, test again.
4. Push to your fork.
5. Open a pull request.

### Tests

Having a wide and comprehensive library of unit-tests and integration tests is of exceeding importance. Contributing tests is widely regarded as a very prestigious contribution (you're making everybody's future work much easier by doing so). Good karma for you. Cookie points. Maybe even a beer if we meet in person :)

Generally tests should be:

- Unitary (as much as possible). I.E. should test as much as possible only one function/method/class. That's the very definition of unit tests. Integration tests are interesting too obviously, but require more time to maintain since they have a higher probability of breaking.
- Short running. No hard numbers here, but if your one test doubles the time it takes for everybody to run them, it's probably an indication that you're doing it wrong.

In a similar way to code, pull requests will be reviewed before pulling (obviously), and we encourage discussion via code review (everybody learns something this way) or IRC discussions.

### Running the tests

To run the tests simply execute `runtests.sh` from your shell. To make sure you have the correct environment you should also provide the `--rebuild-env` flag, but since that makes running the test suite slower, it's disabled by default. You can also see all flags using `--help`.

## 4.1.4 Contributing Documentation

Perhaps considered "boring" by hard-core coders, documentation is sometimes even more important than code! This is what brings fresh blood to a project, and serves as a reference for old timers. On top of this, documentation is the one area where less technical people can help most - you just need to write a semi-decent English. People need to understand you. We don't care about style or correctness.

Documentation should be:

- We use [Sphinx/restructuredText](#). So obviously this is the format you should use :) File extensions should be `.rst`.
- Written in English. We can discuss how it would bring more people to the project to have a Klingon translation or anything, but that's a problem we will ask ourselves when we already have a good documentation in English.
- Accessible. You should assume the reader to be moderately familiar with Python and Django, but not anything else. Link to documentation of libraries you use, for example, even if they are "obvious" to you (South is the first example that comes to mind - it's obvious to any Django programmer, but not to any newbie at all). A brief description of what it does is also welcome.

Pulling of documentation is pretty fast and painless. Usually somebody goes over your text and merges it, since there are no "breaks" and that github parses rst files automagically it's really convenient to work with.

Also, contributing to the documentation will earn you great respect from the core developers. You get good karma just like a test contributor, but you get double cookie points. Seriously. You rock.

## Section style

We use Python documentation conventions fo section marking:

- # with underline, for parts
- \* with underline, for chapters
- =, for sections
- -, for subsections
- ^, for subsubsections
- ", for paragraphs

### 4.1.5 Translations

For translators we have a [transifex account](#) where you can translate the `.po` files and don't need to install git or mercurial to be able to contribute. All changes there will be automatically sent to the project.

## 4.2 Indices and tables

- [genindex](#)
- [modindex](#)
- [search](#)



**c**

`cms.plugin_base`, 47

**m**

`menus.base`, 48



**A**

admin\_preview (cms.plugin\_base.CMSPluginBase attribute), 47

**C**

change\_form\_template  
(cms.plugin\_base.CMSPluginBase attribute), 47

cms.plugin\_base (module), 47

CMSPluginBase (class in cms.plugin\_base), 47

CMSPluginBase.PluginMedia (class in cms.plugin\_base), 48

css (cms.plugin\_base.CMSPluginBase.PluginMedia attribute), 48

**F**

form (cms.plugin\_base.CMSPluginBase attribute), 47

**I**

icon\_alt() (cms.plugin\_base.CMSPluginBase method), 48

icon\_src() (cms.plugin\_base.CMSPluginBase method), 48

**J**

js (cms.plugin\_base.CMSPluginBase.PluginMedia attribute), 48

**M**

menus.base (module), 48

model (cms.plugin\_base.CMSPluginBase attribute), 47

module (cms.plugin\_base.CMSPluginBase attribute), 48

**N**

name (cms.plugin\_base.CMSPluginBase attribute), 48

NavigationNode (class in menus.base), 48

**R**

render() (cms.plugin\_base.CMSPluginBase method), 48

render\_plugin (cms.plugin\_base.CMSPluginBase attribute), 48

render\_template (cms.plugin\_base.CMSPluginBase attribute), 48

**T**

text\_enabled (cms.plugin\_base.CMSPluginBase attribute), 48