djangocms Documentation

Release

2009, Patrick Lauber

January 06, 2012

CONTENTS

This document refers to version

CHAPTER

ONE

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 1.2.5 (or a 1.3.x release).
- South 0.7.2 or higher
- PIL 1.1.6 or higher
- django-classy-tags 0.3.4.1 or higher
- django-mptt 0.4.2 or higher
- django-sekizai 0.4.2 or higher
- html5lib 0.90 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, Django, django-mptt django-classy-tags, django-sekizai, south and html5lib will be installed automatically.

Recommended

- django-filer with its django CMS plugins, file and image management application to use instead of some core plugins
- django-reversion 1.4, to support versions of your content

On Ubuntu

Warning: The instructions here install certain packages, such as PIL, Django, South and django CMS globally, which is not recommended. We recommend you use virtualenv to use instead. If you chose to do so, install Django, django CMS and South inside a virtualenv.

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==1.3 django-cms south
```

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*.

Note: This will install Django version 1.3 for use with the CMS. While later versions of Django (such as 1.4) are know to work for some people, it is NOT recommended and will not be supported until further notice (and tests)

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 Upgrading a django CMS installation

1.2.1 2.2 Release

2.2 release notes (IN DEVELOPMENT)

What's new in 2.2

`django-mptt` now a proper dependency django-mptt is now used as a proper dependency and is no longer shipped with the django CMS. This solves the version conflict issues many people had when trying to use the django CMS together with other Django apps that require django-mptt. django CMS 2.2 requires django-mptt 0.4.2 or higher.

Warning: Please remove the old `mptt` package from your Python site-packages directory before upgrading. The `setup.py` file will install the django-mptt package as an external dependency!

Django 1.3 support The django CMS 2.2 supports both Django 1.2.5 and Django 1.3.

View permissions You can now give view permissions for django CMS pages to groups and users.

Backwards incompatible changes

`django-sekizai` instead of PluginMedia Due to the sorry state of the old plugin media framework, it has been dropped in favor of the more stable and more flexible django-sekizai, which is a new dependency for the django CMS 2.2.

The following methods and properties of `cms.plugins_base.CMSPluginBase` are affected:

- `cms.plugins_base.CMSPluginBase.PluginMedia`
- `cms.plugins_base.CMSPluginBase.pluginmedia`
- `cms.plugins_base.CMSPluginBase.get_plugin_media()`

Accessing those attribtues or methods will raise a `cms.exceptions.Deprecated` error.

The `cms.middleware.media.PlaceholderMediaMiddleware` middleware was also removed in this process and is therefore no longer required, however you now require to have the `'sekizai.context_processors.sekizai'` context processor in your **:setting:‘django:TEMPLATE_CONTEXT_PROCESSORS‘** setting.

All templates in **:setting:‘CMS_TEMPLATES‘** must contain at least the `js` and `css` sekizai namespaces.

Please refer to the documentation on *Handling media* in custom CMS plugins and the django-sekizai documentation for more information.

Toolbar must be enabled explicitly in templates The toolbar no longer hacks itself into responses in the middleware, but rather has to be enabled explictely using the `{% cms_toolbar %}` template tag from the `cms_tags` template tag library in your templates. The template tag should be placed somewhere within the body of the HTML (within `<body>...</body>`).

This solves issues people where having with the toolbar showing up in places it shouldn't.

Static files moved to /static/ The static files (css/javascript/images) were moved to from `/media/` to `/static/` to work with the new `django.contrib.staticfiles` app in Django 1.3. This means you will have to make sure you serve static files as well as media files on your server.

Features deprecated in 2.2

`django-dbgettext` support The django-dbgettext support has been fully dropped in 2.2 in favor of the built-in mechanisms to achieve multilinguality.

1.3 Introductory Tutorial

This guide assumes your machine meets the requirements outlined in the *Installation* section of this documentation.

1.3.1 Getting help

Should you run into trouble and can't figure out how to solve it yourself, you can get help from either our mailinglist or IRC channel `#django-cms` on the `irc.freenode.net` network.

1.3.2 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.

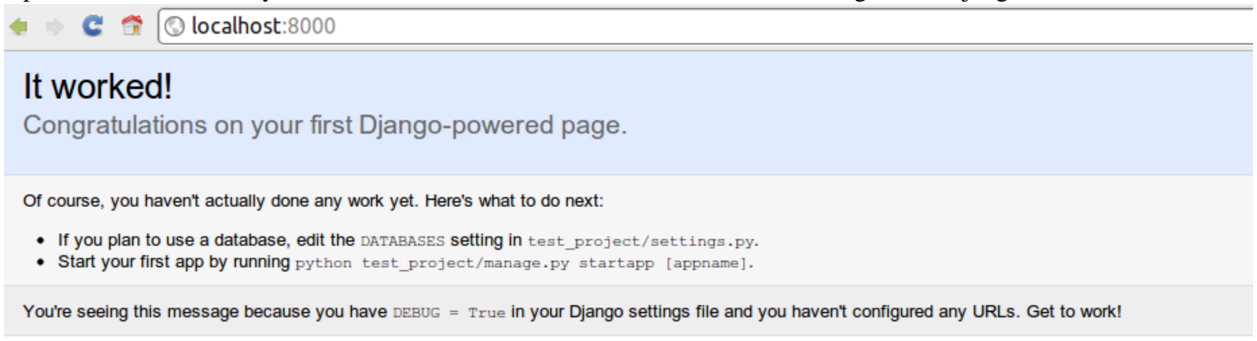
Starting your Django project

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.



It worked!
Congratulations on your first Django-powered page.

Of course, you haven't actually done any work yet. Here's what to do next:

- If you plan to use a database, edit the `DATABASES` setting in `test_project/settings.py`.
- Start your first app by running `python test_project/manage.py startapp [appname]`.

You're seeing this message because you have `DEBUG = True` in your Django settings file and you haven't configured any URLs. Get to work!

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 **:setting:'django:INSTALLED_APPS'** which enable django CMS and required or highly recommended applications/libraries):

- `'cms'`, django CMS itself
- `'mptt'`, utilities for implementing a modified pre-order traversal tree
- `'menus'`, helper for model independent hierarchical website navigation
- `'south'`, intelligent schema and data migrations
- `'sekizai'`, for javascript and css management

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

- `'cms.plugins.file'`
- `'cms.plugins.flash'`
- `'cms.plugins.googlemap'`
- `'cms.plugins.link'`
- `'cms.plugins.picture'`
- `'cms.plugins.snippet'`
- `'cms.plugins.teaser'`
- `'cms.plugins.text'`
- `'cms.plugins.video'`
- `'cms.plugins.twitter'`

They are described in more detail in chapter *Plugins reference*. There is even more plugins available on django CMS extensions page.

Further, make sure you uncomment (enable) `'django.contrib.admin'`

You might consider using django-filer with django CMS plugin and its components instead of `cms.plugins.file`, `cms.plugins.picture`, `cms.plugins.teaser` and `cms.plugins.video` core plugins. In this case you should not add them to **:setting:'django:INSTALLED_APPS'** but add those instead:

- `'filer'`
- `'cmsplugin_filer_file'`
- `'cmsplugin_filer_folder'`
- `'cmsplugin_filer_image'`
- `'cmsplugin_filer_teaser'`
- `'cmsplugin_filer_video'`

If you opt for core plugins you should take care that directory to which **:setting:‘CMS_PAGE_MEDIA_PATH‘** setting points (by default `cms_page_media/` relative to **:setting:‘django:MEDIA_ROOT‘**) is writable by the user under which Django will be running. If you have opted for django-filer then similar requirement exists based on its configuration.

If you want versioning of your content you should also install django-reversion and add it to **:setting:‘django:INSTALLED_APPS‘**:

- `'reversion'`

You need to add the django CMS middlewares to your **:setting:‘django: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.multilingual.MultilingualURLMiddleware',
    'cms.middleware.page.CurrentPageMiddleware',
    'cms.middleware.user.CurrentUserMiddleware',
    'cms.middleware.toolbar.ToolbarMiddleware',
)
```

You need at least the following **:setting:‘django:TEMPLATE_CONTEXT_PROCESSORS‘**:

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

Note: This setting will be missing from automatically generated Django settings files, so you will have to add it.

Point your **:setting:‘django:STATIC_ROOT‘** to where the static files should live (that is, your images, CSS files, Javascript files...):

```
STATIC_ROOT = os.path.join(PROJECT_PATH, "static")
STATIC_URL = "/static/"
ADMIN_MEDIA_PREFIX = "/static/admin/"
```

For uploaded files, you will need to set up the **:setting:‘django:MEDIA_ROOT‘** setting:

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

Note: Please make sure both the `static` and `media` subfolder exist in your project and are writable.

Now add a little magic to the **:setting:‘django: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 **:setting:'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 **:setting:'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 **:setting:'django: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 **:setting:'django:DATABASES'** setting should look like:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(PROJECT_PATH, '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('',
    url(r'^media/(?P<path>.*)$', 'django.views.static.serve',
        {'document_root': settings.MEDIA_ROOT, 'show_indexes': True}),
    url(r'', include('django.contrib.staticfiles.urls')),
) + urlpatterns
```

1.3.3 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 **:setting:‘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 sekizai_tags %}
<html>
  <head>
      {% render_block "css" %}
  </head>
  <body>
      {% placeholder base_content %}
      {% block base_content%}{% endblock %}
      {% render_block "js" %}
  </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".

Static files handling with sekizai

The django CMS handles media files (css stylesheets and javascript files) required by CMS plugins using django-sekizai. This requires you to define at least two sekizai namespaces in your templates: `js` and `css`. You can do so using the `render_block` template tag from the `sekizai_tags` template tag libary. It is highly recommended to put the `{% render_block "css" %}` tag as last thing before the closing `</head>` HTML tag and the `{% render_block "js" %}` tag as the last thing before the closing `</body>` HTML tag.

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.

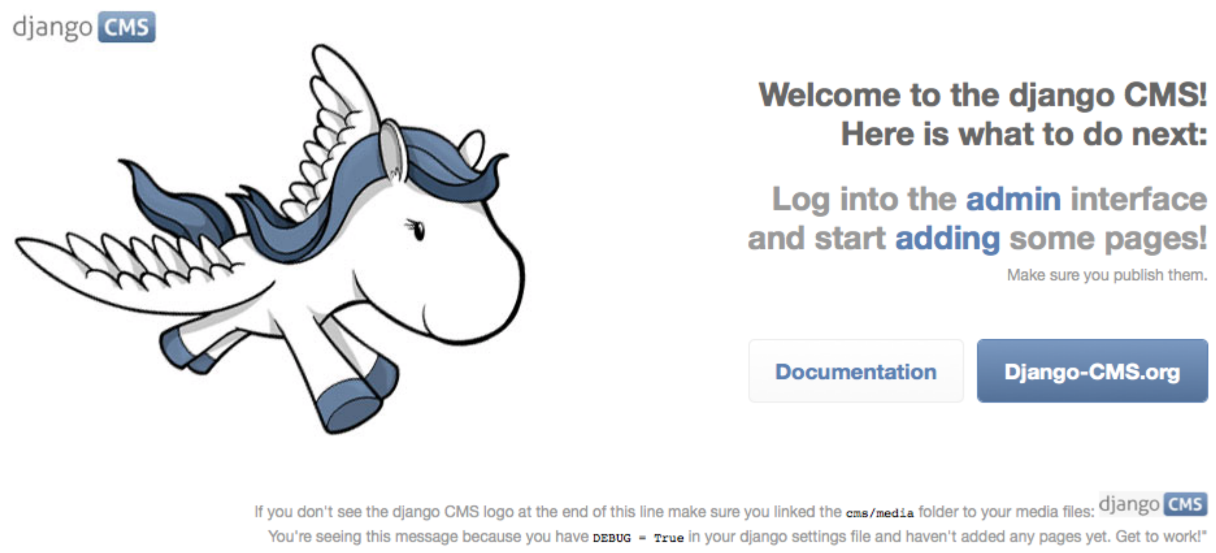
Upgrade

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.



Welcome to the django CMS!
Here is what to do next:

Log into the admin interface and start adding some pages!

Make sure you publish them.

If you don't see the django CMS logo at the end of this line make sure you linked the `cms/media` folder to your media files:

You're seeing this message because you have `DEBUG = True` in your django settings file and haven't added any pages yet. Get to work!"

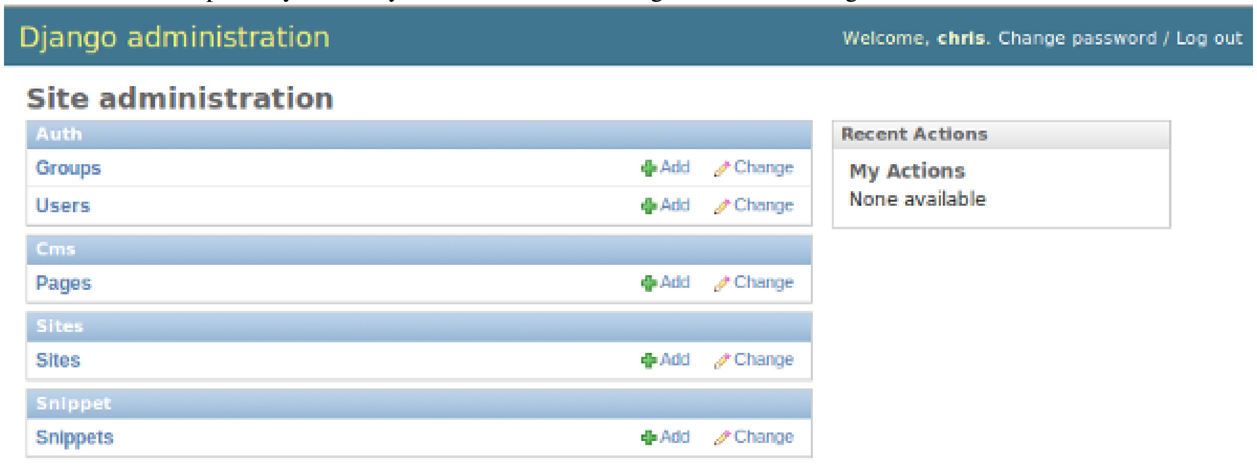
Head over to the *admin panel <http://127.0.0.1:8000/admin/>* and log in with the user you created during the database setup.

To deploy your django CMS project on a production webserver, please refer to the Django documentation.

1.3.4 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:



Django administration

Welcome, **chris**. Change password / Log out

Site administration

Auth		
Groups	Add	Change
Users	Add	Change
Cms		
Pages	Add	Change
Sites		
Sites	Add	Change
Snippet		
Snippets	Add	Change

Recent Actions

My Actions

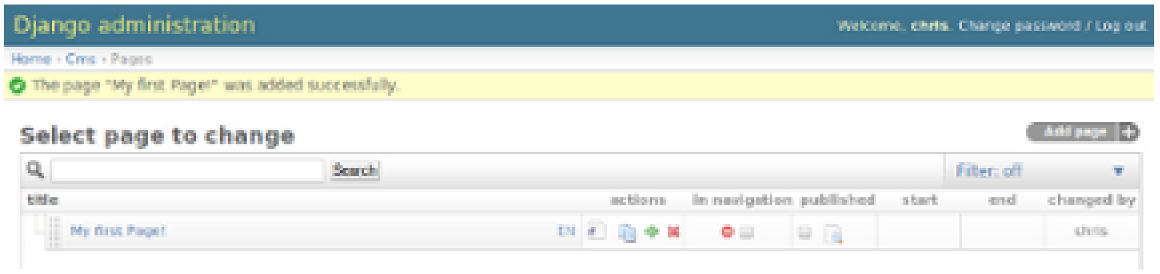
None available

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.



Django administration

Welcome, **chris**. Change password / Log out

Home › Cms › Pages

The page "My first Page!" was added successfully.

Select page to change

title		actions	in navigation	published	start	end	changed by
My first Page!	EN						chris

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)

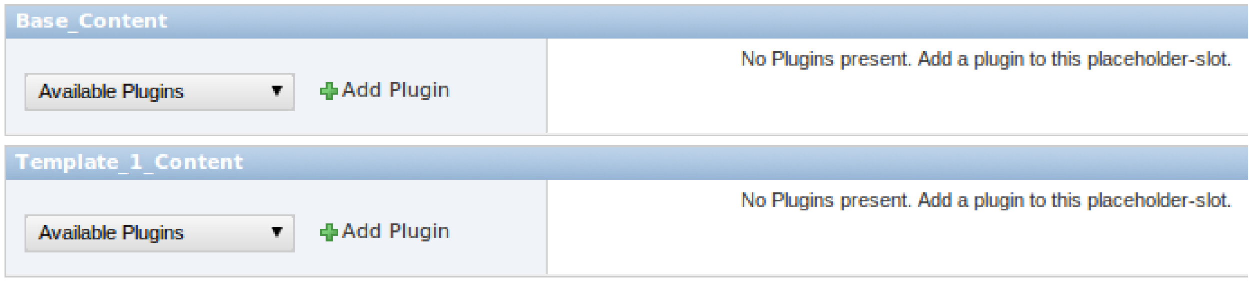
Adding content to a page

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 interfaces shows you theses placeholders as sub menus:



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

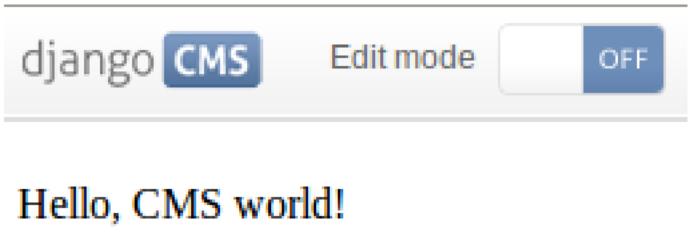
No Plugins present. Add a plugin to this placeholder-slot.

Scroll down the "Available plugins" drop-down list. This displays the plugins you added to your **:setting:'django: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!

1.4 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.

1.4.1 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 **:setting:‘django:INSTALLED_APPS‘** in your `settings.py` file.

1.4.2 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.

1.4.3 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.

1.5 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.

1.5.1 Required Settings

CMS_TEMPLATES

Default: `()` (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')),
)
```

Note: All templates defined in **:setting:‘CMS_TEMPLATES‘** must contain at least the `js` and `css` sekizai namespaces, for more information, see *Static files handling with sekizai*.

1.5.2 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 **:setting:'django: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`.

1.5.3 Editor configuration

The Wymeditor from `cms.plugins.text` plugin can take the same configuration as vanilla Wymeditor. Therefore you will need to learn how to configure that. The best way to understand this is to head over to Wymeditor examples page After understand how Wymeditor works.

The `cms.plugins.text` plugin exposes several variables named WYM_* that correspond to the wym configuration. The simplest way to get started with this is to go to `cms/plugins/text/settings.py` and copy over the WYM_* variables and you will realize they match one to one to Wymeditor's.

Currently the following variables are available:

- `WYM_TOOLS`

- WYM_CONTAINERS
- WYM_CLASSES
- WYM_STYLES
- WYM_STYLESHEET

1.5.4 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 **:setting:'django:LANGUAGES'**

Defines the languages available in django CMS.

Example:

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

Note: Make sure you only define languages which are also in **:setting:'django: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 **:setting:‘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 **:setting:‘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")
```

1.5.5 Media Settings

CMS_MEDIA_PATH

default: `cms/`

The path from **:setting:‘django:MEDIA_ROOT‘** to the media files located in `cms/media/`

CMS_MEDIA_ROOT

Default: **:setting:‘django:MEDIA_ROOT‘** + **:setting:‘CMS_MEDIA_PATH‘**

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

CMS_MEDIA_URL

default: **:setting:‘django:MEDIA_URL‘** + **:setting:‘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.

You should take care that the directory to which it points is writable by the user under which Django will be running.

1.5.6 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.redirects` 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.

1.5.7 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_PUBLIC_FOR

Default: `all`

Decides if pages without any view restrictions are public by default, or staff only. Possible values are `all` and `staff`.

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.

Note: When switching this value to `True` on an existing site, you have to run the `cms moderator on` command to make the required database changes.

CMS_SHOW_START_DATE & CMS_SHOW_END_DATE

Default: `False` for both

This adds two new `DateTimeField` 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_CACHE_DURATIONS

This dictionary carries the various cache duration settings.

`'content'`

Default: `60`

Cache expiration (in seconds) for **:ttag:'show_placeholder'** and **:ttag:'page_url'** template tags.

Note: This settings was previously called **:setting:'CMS_CONTENT_CACHE_DURATION'**

`'menus'`

Default: `3600`

Cache expiration (in seconds) for the menu tree.

Note: This settings was previously called **:setting:'MENU_CACHE_DURATION'**

`'permissions'`

Default: `3600`

Cache expiration (in seconds) for view and other permissions.

CMS_CACHE_PREFIX

Default: `cms-`

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'
```

Note: Django 1.3 introduced a site-wide cache key prefix. See Django's own docs on *cache key prefixing*

1.6 Navigation

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

- **:ttag:'show_menu'**
- **:ttag:'show_menu_below_id'**
- **:ttag:'show_sub_menu'**
- **:ttag:'show_breadcrumb'**

To use any of these templatetags, you need to have `{% load menu_tags %}` in your template before the line on which you call the templatetag.

Note: Please note that menus were originally implemented to be application-independent and as such, live in the `menus` application instead of the `cms` application.

1.6.1 show_menu

:ttag:'{% 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.

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

You can supply a `template` parameter to the tag.

Some Examples

Complete navigation (as a nested list):

```
{% load 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" %}
```

1.6.2 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 **:ttag:'show_menu'**:

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

1.6.3 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>
```

1.6.4 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.

1.6.5 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, without any protocol, domain or port.

```
{{ 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".

1.6.6 Soft Roots

What Soft Roots do

A *soft root* is a page that acts as the root for a menu navigation tree.

Typically, this will be a page that is the root of a significant new section on your site.

When the *soft root* feature is enabled, the navigation menu for any page will start at the nearest *soft root*, rather than at the real root of the site's page hierarchy.

This feature is useful when your site has deep page hierarchies (and therefore multiple levels in its navigation trees). In such a case, you usually don't want to present site visitors with deep menus of nested items.

For example, you're on the page "Introduction to Bleeding", so the menu might look like this:

- **School of Medicine**
    - Medical Education

  - **Departments**
    - Department of Lorem Ipsum
    - Department of Donec Imperdiet
    - Department of Cras Eros
    - **Department of Mediaeval Surgery**
      - Theory
      - **Cures**
        - **Bleeding**
          - Introduction to Bleeding <this is the current page>
          - Bleeding - the scientific evidence
          - Cleaning up the mess
        - Cupping
        - Leaches
        - Maggots
      - Techniques
      - Instruments
    - Department of Curabitur a Purus
    - Department of Sed Accumsan
    - Department of Etiam
  - Research
  - Administration
  - Contact us
  - Impressum

which is frankly overwhelming.

By making “Department of Mediaeval Surgery” a *soft root*, the menu becomes much more manageable:

- **Department of Mediaeval Surgery**
  - Theory
  - **Cures**
    - **Bleeding**
      - Introduction to Bleeding <current page>
      - Bleeding - the scientific evidence
      - Cleaning up the mess
    - Cupping
    - Leaches
    - Maggots
  - Techniques

– Instruments

Using Soft Roots

To enable the feature, `settings.py` requires:

CMS_SOFTROOT = True

Mark a page as *soft root* in the 'Advanced' tab of the its settings in the admin interface.

1.6.7 Modifying & Extending the menu

Please refer to the *App Integration* documentation

1.7 Plugins reference

1.7.1 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 **:setting:'django:INSTALLED_APPS'** setting in your project's `settings.py` file:

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

You should take care that the directory defined by the configuration setting **:setting:'CMS_PAGE_MEDIA_PATH'** (by default `cms_page_media/` relative to **:setting:'django:MEDIA_ROOT'**) is writable by the user under which django will be running.

You might consider using django-filer with django CMS plugin and its `cmsplugin_filer_file` component instead.

Warning: The builtin file plugin does only work with local storages. If you need more advanced solutions, please look at alternative file plugins for the django CMS, such as django-filer.

1.7.2 Flash

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

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

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

1.7.3 GoogleMap

Displays a map of an address on your page.

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

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

1.7.4 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 **:setting:‘django:INSTALLED_APPS‘** setting in your project’s `settings.py` file:

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

Note: As of version 2.2, the link plugin no longer verifies the existance of link targets.

1.7.5 Picture

Displays a picture in a page.

For installation be sure you have the following in the **:setting:‘django: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" %}
    <img src="{% thumbnail picture.image.name 484x1500 upscale %}" {% if picture.alt %}alt="{{ pictu
{% endifequal %}
{% ifequal placeholder "teaser" %}
```

```
    <img src="{% thumbnail picture.image.name 484x1500 upscale %}" {% if picture.alt %}alt="{{ pictu
{% endifequal %}
{% if picture.url %}</a>{% endif %}

{% endspaceless %}
```

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

You should take care that the directory defined by the configuration setting **:setting:‘CMS_PAGE_MEDIA_PATH‘** (by default `cms_page_media/` relative to **:setting:‘django:MEDIA_ROOT‘**) is writable by the user under which django will be running.

Note: For more advanced use cases where you would like to upload your media to a central location, consider using django-filer with django CMS plugin and its `cmsplugin_filer_video` component instead.

1.7.6 Snippet

Renders a HTML snippet from a HTML file in your templates directories or a snippet given via direct input.

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

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

Note: This plugin should mainly be used during development to quickly test HTML snippets.

1.7.7 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 **:setting:‘django:INSTALLED_APPS‘** settings in your project's `settings.py` file:

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

You should take care that the directory defined by the configuration setting **:setting:‘CMS_PAGE_MEDIA_PATH‘** (by default `cms_page_media/` relative to **:setting:‘django:MEDIA_ROOT‘**) is writable by the user under which django will be running.

Note: For more advanced use cases where you would like to upload your media to a central location, consider using django-filer with django CMS plugin and its `cmsplugin_filer_video` component instead.

1.7.8 Text

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

- `cms.plugins.link`
- `cms.plugins.picture`
- `cms.plugins.file`
- `cms.plugins.snippet`

The current editor is Wymeditor. If you want to use TinyMce you need to install django-tinymce. If `tinymce` is in your **:setting:'django: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
```

Note: When using django-tinymce, you also need to configure it. See the django-tinymce docs for more information.

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

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

1.7.9 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 **:setting:'django: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_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")

You should take care that the directory defined by the configuration setting **:setting:'CMS_PAGE_MEDIA_PATH'** (by default cms_page_media/ relative to **:setting:'django:MEDIA_ROOT'**) is writable by the user under which django will be running.

Note: For more advanced use cases where you would like to upload your media to a central location, consider using django-filer with django CMS plugin and its cmsplugin_filer_video component instead.

1.7.10 Twitter

Displays the last number of post of a twitter user.

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

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

Note: Since avatars are not guaranteed to be available over SSL (HTTPS), by default the Twitter plugin does not use avatars on secure sites.

1.7.11 Inherit

Displays all plugins of another page or another 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 **:setting:'django: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.

CHAPTER

TWO

ADVANCED

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 **:setting:'django:MIDDLEWARE_CLASSES'** setting.

Note: This middleware must be put before `cms.middleware.page.CurrentPageMiddleware`

Example:

```
MIDDLEWARE_CLASSES = (
    ...
    'cms.middleware.multilingual.MultilingualURLMiddleware',
    'cms.middleware.user.CurrentUserMiddleware',
    'cms.middleware.page.CurrentPageMiddleware',
    'cms.middleware.toolbar.ToolbarMiddleware'
    ...
)
```

2.1.2 Language Chooser

The **:ttag:'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()` method:

```
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()` method to the `set_language_chooser` function:

```
from menus.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 then 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 `cms.middleware.multilingual.MultilingualURLMiddleware` must be enabled.

simple_language_changer

If the URLs of your views don't actually change besides the language prefix, you can use the `menus.utils.simple_language_changer()` view decorator, instead of manually using *set_language_changer*:

```
from menus.utils import simple_language_changer

@simple_language_changer
def get_prodcut(request, slug):
    # ...
```

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 **:setting:'CMS_HIDE_UNTRANSLATED'** to `False` in your `settings.py` all pages will be displayed in all languages even if they are not translated yet.

If **:setting:‘CMS_HIDE_UNTRANSLATED‘** is `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 **:setting:‘django: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 Template Tags

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 %}` and `{% 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 **:ttag:'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 none of the placeholders on parent pages have plugins that generate content:

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

See also the **:setting:'CMS_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:

- `str`: interpreted as the `reverse_id` field of the desired page, which can be set in the "Advanced" section when editing a page.
- `int`: 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 **:ttag:'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 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>
```

2.3.5 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 %}
```

2.3.6 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" %}
```

2.3.7 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>
```

2.3.8 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>
```

2.3.9 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 %}
```

2.3.10 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*.

2.3.11 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*.

2.3.12 cms_toolbar

The `cms_toolbar` templatetag will add the needed css and javascript to the sekizai blocks in the base template. The templatetag should be placed somewhere within the body of the HTML (within `<body>...</body>`).

Example:

```
<body>
{% cms_toolbar %}
...
```

2.4 Command Line Interface

You can invoke the django CMS command line interface using the `cms` Django command:

```
python manage.py cms
```

2.4.1 Informational commands

`cms list`

The `list` command is used to display information about your installation.

It has two subcommands:

- `cms list plugins` lists all plugins that are used in your project.
- `cms list apphooks` lists all apphooks that are used in your project.

2.4.2 Plugin and apphook management commands

`cms uninstall`

The `uninstall` subcommand can be used to make an uninstallation of a CMS Plugin or an apphook easier.

It has two subcommands:

- `cms uninstall plugins <plugin name> [<plugin name 2> [...]]` uninstalls one or several plugins by **removing** them from all pages where they are used. Note that the plugin name should be the name of the class that is registered to the django CMS. If you are unsure about the plugin name, use the *cms list* to see a list of installed plugins.
- `cms uninstall apphooks <apphook name> [<apphook name 2> [...]]` uninstalls one or several apphooks by **removing** them from all pages where they are used. Note that the apphook name should be the name of the class that is registered to the django CMS. If you are unsure about the apphook name, use the *cms list* to see a list of installed apphook.

Warning: The uninstall command **permanently deletes** data from your database. You should make a backup of your database before using them!

2.4.3 Moderator commands

`cms moderator`

If you turn **:setting:‘CMS_MODERATOR‘** to `True` on an existing project, you should use the `cms moderator on` command to make the required changes in the database, otherwise you will have problems with invisible pages.

Warning: This command **alters data** in your database. You should make a backup of your database before using it!

CHAPTER

THREE

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. Also, make sure the poll app is in your **:setting:‘django:INSTALLED_APPS‘**.

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 `models.Model`.

Run `manage.py 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>{{ instance.poll.question }}</h1>

<form action="{% url polls.views.vote poll.id %}" method="post">
{% csrf_token %}
{% for choice in instance.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 a `CMSApp`. 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
```

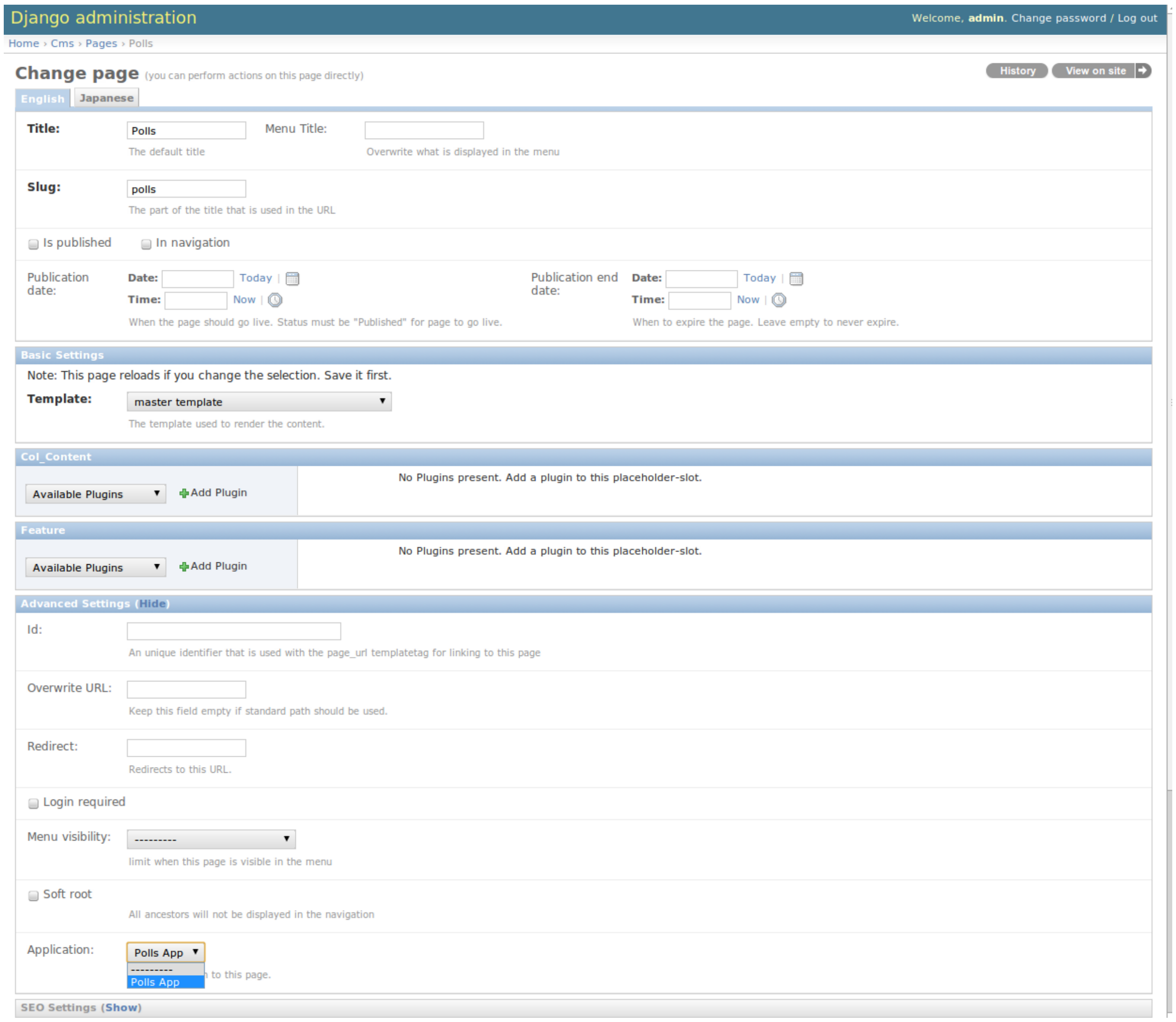
Now remove the inclusion of the polls urls in your main `urls.py` so it looks like this:

```
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'.



Change page (you can perform actions on this page directly)

Title: Polls
The default title

Menu Title:
Overwrite what is displayed in the menu

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

Is published
In navigation

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

Publication end date: Date: Time:
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: master 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: Polls App

Polls App
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"]
    menus = [PollsMenu] # attach a CMSAttachMenu to this apphook.

apphook_pool.register(PollsApp)
```

3.2 Custom Plugins

CMS Plugins are reusable content publishers, that can be inserted into django CMS pages (or indeed into any content that uses django CMS placeholders) in order to publish information automatically, without further intervention.

This means that your published web content, whatever it is, can be kept instantly up-to-date at all times.

It's like magic, but quicker.

Unless you're lucky enough to discover that your needs can be met by the built-in plugins, or by the many available 3rd-party plugins, you'll have to write your own custom CMS Plugin.

Don’t worry though, since writing a CMS Plugin is rather simple.

3.2.1 Why would you need to write a plugin?

A plugin is the most convenient way to integrate content from another Django app into a django CMS page.

For example, suppose you’re developing a site for a record company in django CMS. You might like to have on your site’s home page a “Latest releases” box.

Of course, you could every so often edit that page and update the information. However, a sensible record company will manage its catalogue in Django too, which means Django already knows what this week’s new releases are.

This is an excellent opportunity to make use of that information to make your life easier - all you need to do is create a django CMS plugin that you can insert into your home page, and leave it to do the work of publishing information about the latest releases for you.

Plugins are **reusable**. Perhaps your record company is producing a series of reissues of seminal Swiss punk records; on your site’s page about the series, you could insert the same plugin, configured a little differently, that will publish information about recent new releases in that series.

3.2.2 Overview

A django CMS plugin is fundamentally composed of three things.

- a plugin **editor**, to configure a plugin each time it is deployed
- a plugin **publisher**, to do the automated work of deciding what to publish
- a plugin **template**, to render the information into a web page

These correspond to the familiar with the Model-View-Template scheme:

- the plugin **model** to store its configuration
- the plugin **view** that works out what needs to be displayed
- the plugin **template** to render the information

And so to build your plugin, you’ll make it out of:

- a subclass of `cms.models.pluginmodel.CMSPlugin` to **store the configuration** for your plugin instances
- a subclass of `cms.plugin_base.CMSPluginBase` that **defines the operating logic** of your plugin
- a template that **renders your plugin**

A note about `cms.plugin_base.CMSPluginBase`

`cms.plugin_base.CMSPluginBase` is actually a subclass of `django.contrib.admin.options.ModelAdmin`.

It is its `render()` method that is the plugin’s **view** function.

An aside on models and configuration

The plugin **model**, the subclass of `cms.models.pluginmodel.CMSPlugin`, is actually optional.

You could have a plugin that didn’t need to be configured, because it only ever did one thing.

For example, you could have a plugin that always and only publishes information about the top-selling record of the past seven days. Obviously, this wouldn't be very flexible - you wouldn't be able to use the same plugin to for the best-selling release of the last *month* instead.

Usually, you find that it is useful to be able to configure your plugin, and it will require a model.

3.2.3 The simplest plugin

You may use `python manage.py startapp` to set up the basic layout for you plugin app, alternatively, just add a file called `cms_plugins.py` to an existing Django application.

In there, you place your plugins, in our example the following code:

```
from cms.plugin_base import CMSPluginBase
from cms.plugin_pool import plugin_pool
from cms.models.pluginmodel import CMSPlugin
from django.utils.translation import ugettext_lazy as _

class HelloPlugin(CMSPluginBase):
    model = CMSPlugin
    name = _("Hello Plugin")
    render_template = "hello_plugin.html"

    def render(self, context, instance, placeholder):
        return context

plugin_pool.register_plugin(HelloPlugin)
```

Now we're almost done, all that's left is adding the template. Add the following into the root template directory in a file called `hello_plugin.html`:

```
<h1>Hello {% if request.user.is_authenticated %}{{ request.user.first_name }} {{ request.user.last_na
```

This plugin will now greet the users on your website either by their name if they're logged in, or as Guest if they're not.

Now let's take a closer look at what we did there. The `cms_plugins.py` files are where you should define your subclasses of `cms.plugin_base.CMSPluginBase`, these classes define the different plugins.

There are three required attributes on those classes:

- `model`: The model you wish to use to store information about this plugin, if you do not require any special information, for example configuration, to be stored for your plugins, you may just use `cms.models.pluginmodel.CMSPlugin`. We'll look at that model more closely in a bit.
- `name`: The name of your plugin as displayed in the admin. It is generally good practice to mark this string as translatable using `django.utils.translation.ugettext_lazy()`, however this is optional.
- `render_template`: The template to render this plugin with.

In addition to those three attributes, you must also define a `render()` method on your subclasses. It is specifically this *render* method that is the **view** for your plugin.

That *render* method takes three arguments:

- `context`: The context with which the page is rendered.
- `instance`: The instance of your plugin that is rendered.
- `placeholder`: The name of the placeholder that is rendered.

This method must return a dictionary or an instance of `django.template.Context`, which will be used as context to render the plugin template.

3.2.4 Storing configuration

In many cases, you want to store configuration for your plugin instances, for example if you have a plugin that shows the latest blog posts, you might want to be able to choose the amount of entries shown. Another example would be a gallery plugin, where you want to choose the pictures to show for the plugin.

To do so, you create a Django model by subclassing `cms.models.pluginmodel.CMSPlugin` in the `models.py` of an installed application.

Let's improve our `HelloPlugin` from above by making it configurable what the fallback name for non-authenticated users should be.

In our `models.py` we add following model:

```
from cms.models.pluginmodel import CMSPlugin

from django.db import models

class Hello(CMSPlugin):
    guest_name = models.CharField(max_length=50, default='Guest')
```

If you followed the Django tutorial, this shouldn't look too new to you. The only difference to normal models is that you subclass `cms.models.pluginmodel.CMSPlugin` rather than `django.db.models.base.Model`.

Now we need to change our plugin definition to use this model, so our new `cms_plugins.py` looks like this:

```
from cms.plugin_base import CMSPluginBase
from cms.plugin_pool import plugin_pool
from django.utils.translation import ugettext_lazy as _

from models import Hello

class HelloPlugin(CMSPluginBase):
    model = Hello
    name = _("Hello Plugin")
    render_template = "hello_plugin.html"

    def render(self, context, instance, placeholder):
        context['instance'] = instance
        return context

plugin_pool.register_plugin(HelloPlugin)
```

We changed the `model` attribute to point to our newly created `Hello` model and pass the model instance to the context.

As a last step, we have to update our template to make use of this new configuration:

```
<h1>Hello {% if request.user.is_authenticated %}{{ request.user.first_name }} {{ request.user.last_na
```

The only thing we changed there is that we use the template variable `{{ instance.guest_name }}` instead of the hardcoded `Guest` string in the else clause.

Warning: cms.models.pluginmodel.CMSPlugin subclasses cannot be further subclassed at the moment. In order to make your plugin models reusable, please use abstract base models.

Warning: You cannot name your model fields the same as any installed plugins lower-cased model name, due to the implicit one-to-one relation Django uses for subclassed models. If you use all core plugins, this includes: file, flash, googlemap, link, picture, snippetptr, teaser, twittersearch, twitterrecententries and video.
Additionally, it is *recommended* that you avoid using page as a model field, as it is declared as a property of cms.models.pluginmodel.CMSPlugin, and your plugin will not work as intended in the administration without further work.

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 cms.models.pluginmodel.CMSPlugin.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()
```

3.2.5 Advanced

Plugin form

Since cms.plugin_base.CMSPluginBase extends django.contrib.admin.options.ModelAdmin, you can customize the form for your plugins just as you would customize your admin interfaces.

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

Handling media

If your plugin depends on certain media files, javascript or stylesheets, you can include them from your plugin template using django-sekizai. Your CMS templates are always enforced to have the `css` and `js` sekizai namespaces, therefore those should be used to include the respective files. For more information about django-sekizai, please refer to the django-sekizai documentation.

Sekizai style

To fully harness the power of django-sekizai, it is helpful to have a consistent style on how to use it. Here is a set of conventions that should, but don't necessarily need to, be followed:

- One bit per `addtoblock`. Always include one external CSS or JS file per `addtoblock` or one snippet per `addtoblock`. This is needed so django-sekizai properly detects duplicate files.
- External files should be on one line, with no spaces or newlines between the `addtoblock` tag and the HTML tags.
- When using embedded javascript or CSS, the HTML tags should be on a newline.

A **good** example:

```
{% load sekizai_tags %}

{% addtoblock "js" %}<script type="text/javascript" src="{{ MEDIA_URL }}myplugin/js/myjsfile.js"></sc
{% addtoblock "js" %}<script type="text/javascript" src="{{ MEDIA_URL }}myplugin/js/myotherfile.js"><
{% addtoblock "css" %}<link rel="stylesheet" type="text/css" href="{{ MEDIA_URL }}myplugin/css/astyle
{% addtoblock "js" %}
<script type="text/javascript">
    $(document).ready(function(){
        doSomething();
    });
</script>
{% endaddtoblock %}
```

A **bad** example:

```
{% load sekizai_tags %}

{% addtoblock "js" %}<script type="text/javascript" src="{{ MEDIA_URL }}myplugin/js/myjsfile.js"></sc
<script type="text/javascript" src="{{ MEDIA_URL }}myplugin/js/myotherfile.js"></script>{% endaddtobl
{% addtoblock "css" %}
    <link rel="stylesheet" type="text/css" href="{{ MEDIA_URL }}myplugin/css/astylesheet.css"></scrip
{% endaddtoblock %}
{% addtoblock "js" %}<script type="text/javascript">
    $(document).ready(function(){
        doSomething();
    });
</script>{% endaddtoblock %}
```

Plugin Context Processors

Plugin context processors are callables that modify all plugins' context before rendering. They are enabled using the **:setting:'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:

```
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}
```

Plugin Processors

Plugin processors are callables that modify all plugins' output after rendering. They are enabled using the **:setting:'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: Plugin processors are also applied to plugins embedded in Text plugins (and any custom plugin allowing nested plugins). 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 `div` tags 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" placeho
    '''
    # Plugins not in the main placeholder should remain unchanged
    # Plugins embedded in Text should remain unchanged in order not to break output
    if placeholder.slot != 'main' or (instance._render_meta.text_enabled and instance.parent):
        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: {{ background_co
        # 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') el
            'border_color': instance.border_color if hasattr(instance, 'border_color') else 'lightblu
        })
        # 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 ugettext_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 another 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.

- visible=True

  Whether or not this menu item should be visible.

Additionally, each `NavigationNode` provides a number of methods, which are detailed in the `NavigationNode` API references.

3.3.2 Attach Menus

Classes that extend from `menus.base.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 `cms.menu_bases.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.

Each must have a `get_menu_title()` method, a `get_absolute_url()` method, and a `childrens` list with all of its children inside (the 's' at the end of `childrens` is done on purpose because `children` is already taken by django-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 database 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.

Warning: 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.

Note: If at some point you want to remove this apphook after deleting the cms_app.py there is a cms management command called uninstall apphooks that removes the specified apphook(s) from all pages by name. eg. `manage.py cms uninstall apphooks MyApphook`. To find all names for uninstallable apphooks there is a command for this aswell `manage.py cms list apphooks`.

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 `cms.middleware.multilingual.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 langauge 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
```

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"):
            node = NavigationNode(
                category.name,
                category.get_absolute_url(),
                category.pk,
                category.parent_id
            )
            nodes.append(node)
        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 give your application access to navigation menus.

A modifier can change the properties of existing nodes or rearrange entire menus.

An example use-case

A simple example: you have a news application that publishes pages independently of django CMS. However, you would like to integrate the application into the menu structure of your site, so that at appropriate places a *News* node appears in the navigation menu.

In such a case, a Navigation Modifier is the solution.

How it works

Normally, you'd want to place modifiers in your application's `menu.py`.

To make your modifier available, it then needs to be registered with `menus.menu_pool.menu_pool`.

Now, when a page is loaded and the menu generated, your modifier will be able to inspect and modify its nodes.

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

menu_pool.register_modifier(MyMode)
```

It has a method `modify()` that should return a list of `NavigationNode` instances. `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 two 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 built-in modifier that marks all node levels:

```
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)

menu_pool.register_modifier(Level)
```

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.api

Python APIs for creating CMS contents. This is done in `cms.api` and not on the models and managers, because the direct API via models and managers is slightly counterintuitive for developers. Also the functions defined in this module do sanity checks on arguments.

Warning: None of the functions in this modules do any security or permission checks. They verify their input values to be sane wherever possible, however permission checks should be implemented manually before calling any of these functions.

Functions and constants

`cms.api.`**`VISIBILITY_ALL`**

Used for the `limit_menu_visibility` keyword argument to `create_page()`. Does not limit menu visibility.

`cms.api.`**`VISIBILITY_USERS`**

Used for the `limit_menu_visibility` keyword argument to `create_page()`. Limits menu visibility to authenticated users.

`cms.api.`**`VISIBILITY_STAFF`**

Used for the `limit_menu_visibility` keyword argument to `create_page()`. Limits menu visibility to staff users.

`cms.api.`**`create_page`**(*title*, *template*, *language*, *menu_title=None*, *slug=None*, *apphook=None*, *redirect=None*, *meta_description=None*, *meta_keywords=None*, *created_by='python-api'*, *parent=None*, *publication_date=None*, *publication_end_date=None*, *in_navigation=False*, *soft_root=False*, *reverse_id=None*, *navigation_extenders=None*, *published=False*, *site=None*, *login_required=False*, *limit_visibility_in_menu=VISIBILITY_ALL*, *position="last-child"*)

Creates a `cms.models.pagemodel.Page` instance and returns it. Also creates a `cms.models.titlemodel.Title` instance for the specified language.

Parameters

- **title** (*string*) – Title of the page
- **template** (*string*) – Template to use for this page. Must be in **:setting:'CMS_TEMPLATES'**
- **language** (*string*) – Language code for this page. Must be in **:setting:'django:LANGUAGES'**
- **menu_title** (*string*) – Menu title for this page
- **slug** (*string*) – Slug for the page, by default uses a slugified version of *title*
- **apphook** (string or `cms.app_base.CMSApp` subclass) – Application to hook on this page, must be a valid apphook
- **redirect** (*string*) – URL redirect (only applicable if **:setting:'CMS_REDIRECTS'** is `True`)
- **meta_description** (*string*) – Description of this page for SEO
- **meta_keywords** (*string*) – Keywords for this page for SEO
- **created_by** (string of `django.contrib.auth.models.User` instance) – User that creates this page
- **parent** (`cms.models.pagemodel.Page` instance) – Parent page of this page
- **publication_date** (*datetime*) – Date to publish this page
- **publication_end_date** (*datetime*) – Date to unpublish this page
- **in_navigation** (*boolean*) – Whether this page should be in the navigation or not
- **soft_root** (*boolean*) – Whether this page is a softroot or not
- **reverse_id** (*string*) – Reverse ID of this page (for template tags)
- **navigation_extenders** (*string*) – Menu to attach to this page, must be a valid menu
- **published** (*boolean*) – Whether this page should be published or not
- **site** (`django.contrib.sites.models.Site` instance) – Site to put this page on
- **login_required** (*boolean*) – Whether users must be logged in or not to view this page

- **limit_menu_visibility** (`VISIBILITY_ALL` or `VISIBILITY_USERS` or `VISIBILITY_STAFF`) – Limits visibility of this page in the menu
- **position** (*string*) – Where to insert this node if *parent* is given, must be `'first-child'` or `'last-child'`
- **overwrite_url** (*string*) – Overwritten path for this page

`cms.api.`**`create_title`**(*language*, *title*, *page*, *menu_title=None*, *slug=None*, *apphook=None*, *redirect=None*, *meta_description=None*, *meta_keywords=None*, *parent=None*)

Creates a `cms.models.titlemodel.Title` instance and returns it.

Parameters

- **language** (*string*) – Language code for this page. Must be in **:setting:'django:LANGUAGES'**
- **title** (*string*) – Title of the page
- **page** (`cms.models.pagemodel.Page` instance) – The page for which to create this title
- **menu_title** (*string*) – Menu title for this page
- **slug** (*string*) – Slug for the page, by default uses a slugified version of *title*
- **apphook** (string or `cms.app_base.CMSApp` subclass) – Application to hook on this page, must be a valid apphook
- **redirect** (*string*) – URL redirect (only applicable if **:setting:'CMS_REDIRECTS'** is `True`)
- **meta_description** (*string*) – Description of this page for SEO
- **meta_keywords** (*string*) – Keywords for this page for SEO
- **parent** (`cms.models.pagemodel.Page` instance) – Used for automated slug generation
- **overwrite_url** (*string*) – Overwritten path for this page

`cms.api.`**`add_plugin`**(*placeholder*, *plugin_type*, *language*, *position='last-child'*, ***data*)

Adds a plugin to a placeholder and returns it.

Parameters

- **placeholder** (`cms.models.placeholdermodel.Placeholder` instance) – Placeholder to add the plugin to
- **plugin_type** (string or `cms.plugin_base.CMSPluginBase` subclass, must be a valid plugin) – What type of plugin to add
- **language** (*string*) – Language code for this plugin, must be in **:setting:'django:LANGUAGES'**
- **position** (*string*) – Position to add this plugin to the placeholder, must be a valid django-mptt position
- **data** (*kwargs*) – Data for the plugin type instance

`cms.api.`**`create_page_user`**(*created_by*, *user*, *can_add_page=True*, *can_change_page=True*, *can_delete_page=True*, *can_recover_page=True*, *can_add_pageuser=True*, *can_change_pageuser=True*, *can_delete_pageuser=True*, *can_add_pagepermission=True*, *can_change_pagepermission=True*, *can_delete_pagepermission=True*, *grant_all=False*)

Creates a page user for the user provided and returns that page user.

Parameters

- **created_by** (`django.contrib.auth.models.User` instance) – The user that creates the page user
- **user** (`django.contrib.auth.models.User` instance) – The user to create the page user from
- **can_*** (*boolean*) – Permissions to give the user
- **grant_all** (*boolean*) – Grant all permissions to the user

cms.api.**assign_user_to_page**(*page, user, grant_on=ACCESS_PAGE_AND_DESCENDANTS, can_add=False, can_change=False, can_delete=False, can_change_advanced_settings=False, can_publish=False, can_change_permissions=False, can_move_page=False, can_moderate=False, grant_all=False*)

Assigns a user to a page and gives them some permissions. Returns the `cms.models.permissionmodels.PagePermission` object that gets created.

Parameters

- **page** (`cms.models.pagemodel.Page` instance) – The page to assign the user to
- **user** (`django.contrib.auth.models.User` instance) – The user to assign to the page
- **grant_on** (`cms.models.moderatormodels.ACCESS_PAGE`, `cms.models.moderatormodels.ACCESS_CHILDREN`, `cms.models.moderatormodels.ACCESS_DESCENDANTS` or `cms.models.moderatormodels.ACCESS_PAGE_AND_DESCENDANTS`) – Controls which pages are affected
- **can_*** – Permissions to grant
- **grant_all** (*boolean*) – Grant all permissions to the user

cms.api.**publish_page**(*page, user, approve=False*)

Publishes a page and optionally approves that publication.

Parameters

- **page** (`cms.models.pagemodel.Page` instance) – The page to publish
- **user** (`django.contrib.auth.models.User` instance) – The user that performs this action
- **approve** (*boolean*) – Whether to approve the publication or not

cms.api.**approve_page**(*page, user*)

Approves a page.

Parameters

- **page** (`cms.models.pagemodel.Page` instance) – The page to approve
- **user** (`django.contrib.auth.models.User` instance) – The user that performs this action

Example workflows

Create a page called `'My Page` using the template `'my_template.html'` and add a text plugin with the content `'hello world'`. This is done in English:

```
from cms.api import create_page, add_plugin

page = create_page('My Page', 'my_template.html', 'en')
placeholder = page.placeholders.get(slot='body')
add_plugin(placeholder, 'TextPlugin', 'en', body='hello world')
```

3.4.2 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 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`

3.4.3 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.

`get_descendants`()
Returns a list of all children beneath the current menu item.

`get_ancestors`()
Returns a list of all parent items, excluding the current menu item.

`get_absolute_url`()
Utility method to return the URL associated with this menu item, primarily to follow naming convention asserted by Django.

get_menu_title()
Utility method to return the associated title, using the same naming convention used by `cms.models.pagemodel.Page`.

3.5 Placeholders outside the CMS

Placeholders are special model fields that django CMS 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 for `CMSPlugin` instances, 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 `CMSPlugin`, 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 **:ttag:'render_placeholder'** tag from the `placeholder_tags` template tag library:

```
{% load placeholder_tags %}

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

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

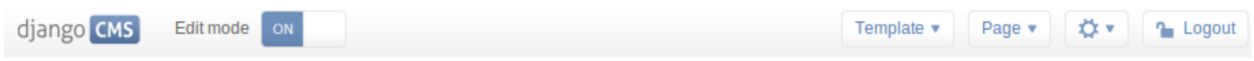
3.5.2 Adding content to a placeholder

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

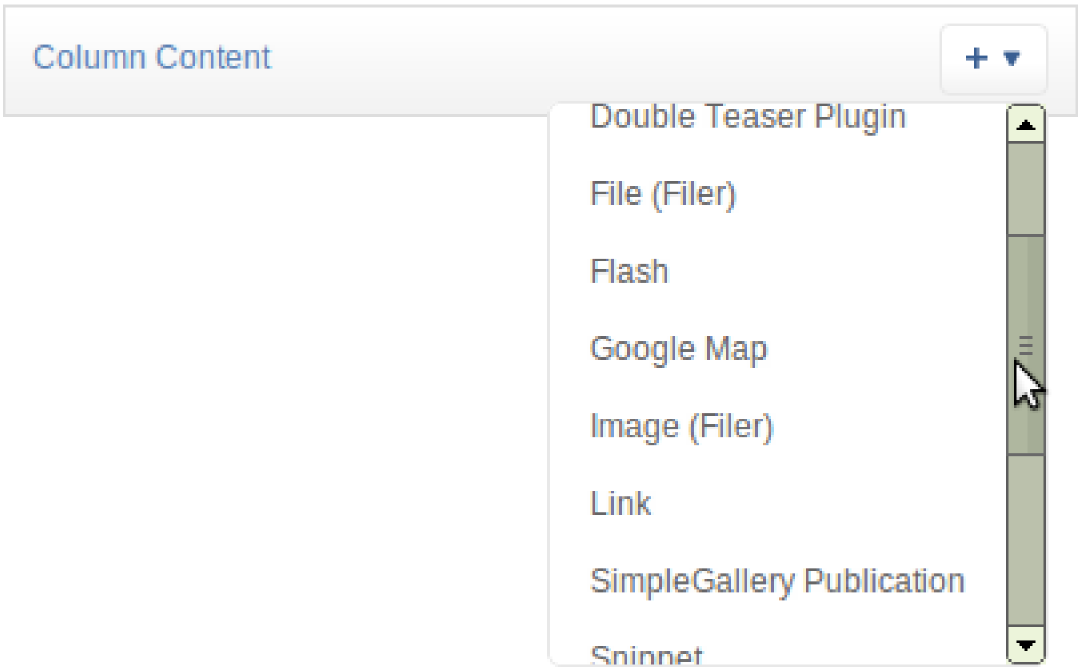
Using the front-end editor

Probably the most simple way to add content to a placeholder, simply visit the page displaying your model (where you put the **:ttag:'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 and 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 screen shot 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 `fieldset`, 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

For powerful full-text search in with the django CMS, we suggest using Haystack together with django-cms-search.

3.7 Form and model fields

3.7.1 Model fields

class `cms.models.fields.`**`PageField`**

This is a foreign key field to the `cms.models.pagemodel.Page` model that defaults to the `cms.forms.fields.PageSelectFormField` form field when rendered in forms. It has the same API as the `django.db.models.fields.related.ForeignKey` but does not require the `othermodel` argument.

3.7.2 Form fields

class `cms.forms.fields.`**`PageSelectFormField`**

Behaves like a `django.forms.models.ModelChoiceField` field for the `cms.models.pagemodel.Page` model, but displays itself as a split field with a select dropdown for the site and one for the page. It also indents the page names based on what level they're on, so that the page select dropdown is easier to use. This takes the same arguments as `django.forms.models.ModelChoiceField`.

CHAPTER

FOUR

CONTRIBUTING TO DJANGO CMS

4.1 Contributing to django CMS

Like every open-source project, django CMS is always looking for motivated individuals to contribute to it's 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 the freenode IRC network 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 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 overline, for parts
- `*` with overline, 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*

PYTHON MODULE INDEX