django cms Documentation

Release 2.4.3-support

Patrick Lauber

June 14, 2016

Contents

This document refers to version 2.4.3-support

CHAPTER 1

Install

Warning: Version 2.4 introduces some significant changes that **require** action if you are upgrading from a previous version. Please refer to *Upgrading from previous versions*

1.1 Installation

This document assumes you are familiar with Python and Django. It 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 1.4.5, 1.5 or higher
- South 0.7.2 or higher
- django-classy-tags 0.3.4.1 or higher
- django-mptt 0.5.2 (strict due to API compatibility issues)
- django-sekizai 0.7 or higher
- html5lib 0.90 or higher
- django-i18nurls (if using django 1.3.X)
- 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

These packages are not *required*, but they provide useful functionality with minimal additional configuration and are well-proven.

File and image handling

- Django Filer for file and image management
- django-filer plugins for django-cms, required to use Django Filer with django-cms

- Pillow (fork of PIL) for image manipulation

Revision management

- django-reversion 1.6.6 (with Django 1.4.5), 1.7 (with Django 1.5) to support versions of your content (If using a different Django version it is a good idea to check the page Compatible-Django-Versions in the django-reversion wiki in order to make sure that the package versions are compatible.)

Note: As of django CMS 2.4, only the most recent 25 published revisions are saved. You can change this behaviour if required with `CMS_MAX_PAGE_PUBLISH_REVERSIONS`. Be aware that saved revisions will cause your database size to increase.

1.1.2 Installing in a virtualenv using pip

Installing inside a virtualenv is the preferred way to install any Django installation. This should work on any platform where python in installed. The first step is to create the virtualenv:

```
#!/bin/sh
sudo pip install --upgrade virtualenv
virtualenv --distribute --no-site-packages env
```

You can switch to your virtualenv at the command line by typing:

```
source env/bin/activate
```

Next, you can install packages one at a time using pip, but we recommend using a requirements.txt file. The following is an example requirements.txt file that can be used with pip to install django-cms and its dependencies:

```
# Bare minimum
django-cms==2.4.1

#These dependencies are brought in by django-cms, but if you want to lock-in their version, speci
Django==1.5.1
django-classy-tags==0.4
South==0.8.1
html5lib==1.0b1
django-mptt==0.5.2
django-sekizai==0.7
six==1.3.0

#Optional, recommended packages
Pillow==2.0.0
django-filer==0.9.4
cmsplugin-filer==0.9.5
django-reversion==1.7
```

for Postgresql you would also add:

```
psycopg2==2.5
```

and install libpq-dev (on Debian-based distro)

for MySQL you would also add:

```
mysql-python==1.2.4
```

and install libmysqlclient-dev (on Debian-based distro)

One example of a script to create a virtualenv Python environment is as follows:

```
#!/bin/sh
env/bin/pip install --download-cache=~/.pip-cache -r requirements.txt
```

1.1.3 Installing globally on Ubuntu

Warning: The instructions here install certain packages, such as Django, South, Pillow and django CMS globally, which is not recommended. We recommend you use virtualenv instead (see above).

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

```
sudo aptitude install python2.6 python-setuptools
sudo easy_install pip
sudo pip install Django==1.5 django-cms south Pillow
```

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

```
sudo aptitude python-psycopg2
```

or

```
sudo aptitude install python-mysql
```

This will install Django, django CMS, South, Pillow, and your database's driver globally.

You have now everything that is needed for you to follow the Introductory Tutorial.

1.1.4 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 systems' respective websites.

To use django CMS efficiently, we recommend:

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

1.2 2.4.3 release notes

1.2.1 What's new in 2.4.3

Fixture loading in Postgres

Placeholder rescan is skipped when loading fixtures.

Fix placeholder primary keys formatting

Fixed a bug when the Django setting USE_THOUSAND_SEPARATOR = True, which caused placeholders primary keys to be formatted according to locale, breaking frontend editing.

Fix show_placeholder in preview mode

`show_placeholder` no longer uses cached content in preview mode.

Other fixes

- Test fixes
- Fix issues in cookies handling
- Fix minor unicode issues
- Fix a missing argument in ModelAdmin
- Fix a bug in WymEditor handling
- Fix bugs in migrations
- Fix bug in language fallback
- Minor documentation fixes

1.3 2.4 release notes

1.3.1 What's new in 2.4

> **Warning:** Upgrading from previous versions
> 2.4 introduces some changes that **require** action if you are upgrading from a previous version.
> You will need to read the sections *Migrations overhaul* and *Added a check command* below.

Introducing Django 1.5 support, dropped support for Django 1.3 and Python 2.5

Django CMS 2.4 introduces Django 1.5 support.

In django CMS 2.4 we dropped support for Django 1.3 and Python 2.5. Django 1.4 and Python 2.6 are now the minimum required versions.

Migrations overhaul

In version 2.4, migrations have been completely rewritten to address issues with newer South releases.

To ease the upgrading process, all the migrations for the *cms* application have been consolidated into a single migration file, *0001_initial.py*.

- migration 0001 is a *real* migration, that gets you to the same point migrations 0001-0036 used to
- the migrations 0002 to 0036 inclusive still exist, but are now all *dummy* migrations
- migrations 0037 and later are *new* migrations

How this affects you

If you're starting with *a new installation*, you don't need to worry about this. Don't even bother reading this section; it's for upgraders.

If you're using version *2.3.2 or newer*, you don't need to worry about this either.

If you're using version *2.3.1 or older*, you will need to run a two-step process.

First, you'll need to upgrade to 2.3.3, to bring your migration history up-to-date with the new scheme. Then you'll need to perform the migrations for 2.4.

For the two-step upgrade process do the following in your project main directory:

```
pip install django-cms==2.3.3
python manage.py syncdb
python manage.py migrate
pip install django-cms==2.4
python manage.py migrate
```

Added delete orphaned plugins command

Added a management command for deleting orphaned plugins from the database.

The command can be run with:

```
manage.py cms delete_orphaned_plugins
```

Please read *cms delete_orphaned_plugins* before using.

Added a check command

Added a management command to check your configuration and environment.

To use this command, simply run:

    manage.py cms check

This replaces the old at-runtime checks.

CMS_MODERATOR

Has been removed since it is no longer in use. From 2.4 onwards, all pages exist in a public and draft version. Users with the `publish_page` permission can publish changes to the public site.

Management command required

To bring a previous version of your site's database up-to-date, you'll need to run `manage.py cms moderator on`. **Never run this command without first checking for orphaned plugins**, using the `cms list plugins` command. If it reports problems, run `manage.py cms delete_orphaned_plugins`. Running `cms moderator` with orphaned plugins will fail and leave bad data in your database. See *cms list* and *cms delete_orphaned_plugins*.

Also, check if all your plugins define a `copy_relations()` method if required. You can do this by running `manage.py cms check` and read the *Presence of "copy_relations"* section. See *Handling Relations* for guidance on this topic.

Added Fix MPTTT Management command

Added a management command for fixing MPTT tree data.

The command can be run with:

```
manage.py cms fix-mptt
```

Removed the MultilingualMiddleware

We removed the MultilingualMiddleware. This removed the very ugly monkey patching of the reverse() function as well. As a benefit we now support localization of urls and Apphook urls with standard django helpers.

For django 1.4 more infos can be found here:

https://docs.djangoproject.com/en/dev/topics/i18n/translation/#internationalization-in-url-patterns

If you are still running django 1.3 you are able to archieve the same functionality with django-i18nurl. It is a backport of the new functionality in django 1.4 and can be found here:

https://github.com/brocaar/django-i18nurls

What you need to do:

- Remove `cms.middleware.multilingual.MultilingualURLMiddleware` from your settings.
- Be sure `django.middleware.locale.LocaleMiddleware` is in your settings, and that it comes after the SessionMiddleware.
- Be sure that the `cms.urls` is included in a `i18n_patterns`:

```
from django.conf.urls.defaults import *
from django.conf.urls.i18n import i18n_patterns
from django.contrib import admin
from django.conf import settings

admin.autodiscover()

urlpatterns = i18n_patterns('',
    url(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
```

- Change your url and reverse calls to language namespaces. We now support the django way of calling other language urls either via `{% language %}` templatetag or via `activate("de")` function call in views.

  Before:

```
{% url "de:myview" %}
```

  After:

```
{% load i18n %}{% language "de" %}
{% url "myview_name" %}
{% endlanguage %}
```

- reverse urls now return the language prefix as well. So maybe there is some code that adds language prefixes. Remove this code.

Added LanguageCookieMiddleware

To fix the behavior of django to determine the language every time from new, when you visit / on a page, this middleware saves the current language in a cookie with every response.

To enable this middleware add the following to your *MIDDLEWARE_CLASSES* setting:

cms.middleware.language.LanguageCookieMiddleware

CMS_LANGUAGES

CMS_LANGUAGES has be overhauled. It is no longer a list of tuples like the `LANGUAGES` settings.

An example explains more than thousand words:

```
CMS_LANGUAGES = {
        1: [
            {
                'code': 'en',
                'name': gettext('English'),
                'fallbacks': ['de', 'fr'],
                'public': True,
                'hide_untranslated': True,
                'redirect_on_fallback':False,
            },
            {
                'code': 'de',
                'name': gettext('Deutsch'),
                'fallbacks': ['en', 'fr'],
                'public': True,
            },
            {
                'code': 'fr',
                'name': gettext('French'),
                'public': False,
            },
        ],
        2: [
            {
                'code': 'nl',
                'name': gettext('Dutch'),
                'public': True,
                'fallbacks': ['en'],
            },
        ],
        'default': {
            'fallbacks': ['en', 'de', 'fr'],
            'redirect_on_fallback':True,
            'public': False,
            'hide_untranslated': False,
        }
    }
```

For more details on what all the parameters mean please refer to the *CMS_LANGUAGES* docs.

The following settings are not needed any more and have been removed:

- *CMS_HIDE_UNTRANSLATED*
- *CMS_LANGUAGE_FALLBACK*
- *CMS_LANGUAGE_CONF*
- *CMS_SITE_LANGUAGES*
- *CMS_FRONTEND_LANGUAGES*

Please remove them from your `settings.py`.

CMS_FLAT_URLS

Was marked deprecated in 2.3 and has now been removed.

Plugins in Plugins

We added the ability to have plugins in plugins. Until now only the TextPlugin supported this. For demonstration purposes we created a MultiColumn Plugin. The possibilities for this are endless. Imagine: StylePlugin,

TablePlugin, GalleryPlugin etc.

The column plugin can be found here:

https://github.com/divio/djangocms-column

At the moment the limitation is that plugins in plugins is only editable in the frontend.

Here is the MultiColumn Plugin as an example:

```
class MultiColumnPlugin(CMSPluginBase):
    model = MultiColumns
    name = _("Multi Columns")
    render_template = "cms/plugins/multi_column.html"
    allow_children = True
    child_classes = ["ColumnPlugin"]
```

There are 2 new properties for plugins:

allow_children

Boolean If set to True it allows adding Plugins.

child_classes

List A List of Plugin Classes that can be added to this plugin. If not provided you can add all plugins that are available in this placeholder.

How to render your child plugins in the template

We introduce a new templatetag in the cms_tags called `{% render_plugin %}` Here is an example of how the MultiColumn plugin uses it:

```
{% load cms_tags %}
<div class="multicolumn">
{% for plugin in instance.child_plugins %}
    {% render_plugin plugin %}
{% endfor %}
</div>
```

As you can see the children are accessible via the plugins children attribute.

New way to handle django CMS settings

If you have code that needs to access django CMS settings (settings prefixed with `CMS_` or `PLACEHOLDER_`) you would have used for example `from django.conf import settings; settings.CMS_TEMPLATES`. This will no longer guarantee to return sane values, instead you should use `cms.utils.conf.get_cms_setting` which takes the name of the setting **without** the `CMS_` prefix as argument and returns the setting.

Example of old, now deprecated style:

```
from django.conf import settings

settings.CMS_TEMPLATES
settings.PLACEHOLDER_FRONTEND_EDITING
```

Should be replaced with the new API:

```
from cms.utils.conf import get_cms_setting

get_cms_setting('TEMPLATES')
get_cms_setting('PLACEHOLDER_FRONTEND_EDITING')
```

Added `cms.constants` module

This release adds the `cms.constants` module which will hold generic django CMS constant values. Currently it only contains `TEMPLATE_INHERITANCE_MAGIC` which used to live in `cms.conf.global_settings` but was moved to the new `cms.constants` module in the settings overhaul mentioned above.

django-reversion integration changes

django-reversion integration has changed. Because of huge databases after some time we introduce some changes to the way revisions are handled for pages.

1. Only publish revisions are saved. All other revisions are deleted when you publish a page.
2. By default only the latest 25 publish revisions are kept. You can change this behavior with the new `CMS_MAX_PAGE_PUBLISH_REVERSIONS` setting.

Changes to the show_sub_menu templatetag

the *`show_sub_menu`* has received two new parameters. The first stays the same and is still: how many levels of menu should be displayed.

The second: `root_level` (default=None), specifies at what level, if any, the menu should root at. For example, if root_level is 0 the menu will start at that level regardless of what level the current page is on.

The third argument: `nephews` (default=100), specifies how many levels of nephews (children of siblings) are shown.

PlaceholderAdmin support i18n

If you use placeholders in other apps or models we now support more than one language out of the box. If you just use the `PlaceholderAdmin` it will display language tabs like the cms. If you use django-hvad it uses the hvad language tabs.

If you want to disable this behavior you can set `render_placeholder_language_tabs = False` on your Admin class that extends PlaceholderAdmin. If you use a custom `change_form_template` be sure to have a look at `cms/templates/admin/placeholders/placeholder/change_form.html` for how to incorporate language tabs.

Added CMS_RAW_ID_USERS

If you have a lot of users (500+) you can set this setting to a number after which admin User fields are displayed in a raw Id field. This improves performance a lot in the admin as it has not to load all the users into the html.

1.3.2 Backwards incompatible changes

New minimum requirements for dependencies

- Django 1.3 and Python 2.5 are no longer supported.

1.3.3 Pending deprecations

- `simple_language_changer` will be removed in version 3.0. A bugfix makes this redundant as every non managed url will behave like this.

1.4 2.3.8 release notes

1.4.1 What's new in 2.3.8

Fixture loading in Postgres

Placeholder rescan is skipped when loading fixtures.

Fix placeholder primary keys formatting

Fixed a bug when the Django setting USE_THOUSAND_SEPARATOR = True, which caused placeholders primary keys to be formatted according to locale, breaking frontend editing.

1.5 2.3.7 release notes

Warning: Upgrading from previous versions
2.3.7 now requires `django-sekizai>=0.7` (up from `0.6.1`) due to a cache-related fix backported from 2.4. The correct `django-sekizai` version should be automatically installed (if not already present), but please do check when upgrading.

1.5.1 What's new in 2.3.7

Cache-related plugin template fix

Plugin templates may suffer for incorrect caching when using django-sekizai (which may cause css and javascript files not to be served to the users). This backported fix resolve this, as the whole context is now saved in cache along with the rendered templates.

Permissions cache performance issue fix backported

When a page is saved and cache is enabled, an explicit cache clear used to be called, causing performance issues when you have thousand of users on your website. This is no longer the case: cache is now versioned and as a result massive cache invalidation is not needed anymore

Fix MPTT Management command backported

Backported from 2.4 a management command for fixing MPTT tree data.

The command can be run with:

```
manage.py cms fix-mptt
```

1.6 2.3.4 release notes

1.6.1 What's new in 2.3.4

WymEditor fixed

2.3.4 fixes a critical issue with WymEditor that prevented it from load it's JavaScript assets correctly.

Moved Norwegian translations

The Norwegian translations are now available as `nb`, which is the new (since 2003) official language code for Norwegian, replacing the older and deprecated `no` code.

If your site runs in Norwegian, you need to change your `LANGUAGES` settings!

Added support for timezones

On Django 1.4, and with `USE_TZ=True` the django CMS now uses timezone aware date and time objects.

Fixed slug clashing

In earlier versions, publishing a page that has the same slug (URL) as another (published) page could lead to errors. Now, when a page which would have the same URL as another (published) page is published, the user is shown an error and they're prompted to change the slug for the page.

Prevent unnamed related names for PlaceholderField

`cms.models.fields.PlaceholderField` no longer allows the related name to be suppressed. Trying to do so will lead to a `ValueError`. This change was done to allow the django CMS to properly check permissions on Placeholder Fields.

Two fixes to page change form

The change form for pages would throw errors if the user editing the page does not have the permission to publish this page. This issue was resolved.

Further the page change form would not correctly pre-popluate the slug field if `DEBUG` was set to `False`. Again, this issue is now resolved.

1.7 2.3.3 release notes

1.7.1 What's new in 2.3.3

Restored Python 2.5 support

2.3.3 restores Python 2.5 suppport for the django CMS.

1.7.2 Pending deprecations

Python 2.5 support will be dropped in django CMS 2.4.

1.8 2.3.2 release notes

1.8.1 What's new in 2.3.2

Google map plugin

Google map plugin now supports width and height fields so that plugin size can be modified in the page admin or frontend editor.

Zoom level is now set via a select field which ensure only legal values are used.

Warning: Due to the above change, *level* field is now marked as *NOT NULL*, and a datamigration has been introduced to modify existing googlemap plugin instance to set the default value if *level* if is *NULL*.

1.9 2.3 release notes

1.9.1 What's new in 2.3

Introducing Django 1.4 support, dropped support for Django 1.2

In django CMS 2.3 we dropped support for Django 1.2. Django 1.3.1 is now the minimum required Django version. Django CMS 2.3 also introduces Django 1.4 support.

Lazy page tree loading in admin

Thanks to the work by Andrew Schoen the page tree in the admin now loads lazily, significantly improving the performance of that view for large sites.

Toolbar isolation

The toolbar JavaScript dependencies should now be properly isolated and no longer pollute the global JavaScript namespace.

Plugin cancel button fixed

The cancel button in plugin change forms no longer saves the changes, but actually cancels.

Tests refactor

Tests can now be run using `setup.py test` or `runtests.py` (the latter should be done in a virtualenv with the proper dependencies installed).

Check `runtests.py -h` for options.

Moving text plugins to different placeholders no longer loses inline plugins

A serious bug where a text plugin with inline plugins would lose all the inline plugins when moved to a different placeholder has been fixed.

Minor improvements

- The `or` clause in the `placeholder` tag now works correctly on non-cms pages.
- The icon source URL for inline plugins for text plugins no longer gets double escaped.
- `PageSelectWidget` correctly orders pages again.
- Fixed the file plugin which was sometimes causing invalid HTML (unclosed `span` tag).
- Migration ordering for plugins improved.
- Internationalized strings in JavaScript now get escaped.

1.9.2 Backwards incompatible changes

New minimum requirements for dependencies

- `django-reversion` must now be at version 1.6
- `django-sekizai` must be at least at version 0.6.1
- `django-mptt` version 0.5.1 or 0.5.2 is required

Registering a list of plugins in the plugin pool

This feature was deprecated in version 2.2 and removed in 2.3. Code like this will not work anymore:

```
plugin_pool.register_plugin([FooPlugin, BarPlugin])
```

Instead, use multiple calls to `register_plugin`:

```
plugin_pool.register_plugin(FooPlugin)
plugin_pool.register_plugin(BarPlugin)
```

1.9.3 Pending deprecations

The `CMS_FLAT_URLS` setting is deprecated and will be removed in version 2.4. The moderation feature (`CMS_MODERATOR = True`) will be deprecated in 2.4 and replaced with a simpler way of handling unpublished changes.

1.10 2.2 release notes

1.10.1 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 were experiencing when trying to use the django CMS together with other Django apps that require django-mptt. django CMS 2.2 requires django-mptt 0.5.1.

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.

1.10.2 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 attributes 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 are now required to have the `'sekizai.context_processors.sekizai'` context processor in your `TEMPLATE_CONTEXT_PROCESSORS` setting.

All templates in *`CMS_TEMPLATES`* must at least contain 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 explicitly 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 were having with the toolbar showing up in places it shouldn't have.

Static files moved to /static/

The static files (css/javascript/images) were moved 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.

> **Warning:** If you use Django 1.2.x you will not have a `django.contrib.staticfiles` app. Instead you need the django-staticfiles backport.

1.10.3 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.

1.11 Upgrading from 2.1.x and Django 1.2.x

1.11.1 Upgrading dependencies

Upgrade both your version of django CMS and Django by running the following commands.

```
pip install --upgrade django-cms==2.2 django==1.3.1
```

If you are using django-reversion make sure to have at least version 1.4 installed

```
pip install --upgrade django-reversion==1.4
```

Also, make sure that django-mptt stays at a version compatible with django CMS

```
pip install --upgrade django-mptt==0.5.1
```

1.11.2 Updates to `settings.py`

The following changes will need to be made in your settings.py file:

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

Note: These are not django CMS settings. Refer to the Django documentation on staticfiles for more information.

Note: Please make sure the `static` subfolder exists in your project and is writable.

Note: PROJECT_PATH is the absolute path to your project. See *Installing and configuring django CMS in your Django project* for instructions on how to set PROJECT_PATH.

Remove the following from `TEMPLATE_CONTEXT_PROCESSORS`:

```
django.core.context_processors.auth
```

Add the following to `TEMPLATE_CONTEXT_PROCESSORS`:

```
django.contrib.auth.context_processors.auth
django.core.context_processors.static
sekizai.context_processors.sekizai
```

Remove the following from `MIDDLEWARE_CLASSES`:

```
cms.middleware.media.PlaceholderMediaMiddleware
```

Remove the following from `INSTALLED_APPS`:

```
publisher
```

Add the following to `INSTALLED_APPS`:

```
sekizai
django.contrib.staticfiles
```

1.11.3 Template Updates

Make sure to add sekizai tags and `cms_toolbar` to your CMS templates.

Note: `cms_toolbar` is only needed if you wish to use the front-end editing. See *Backwards incompatible changes* for more information

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

```
{% load cms_tags sekizai_tags %}
<html>
  <head>
      {% render_block "css" %}
  </head>
  <body>
      {% cms_toolbar %}
      {% placeholder base_content %}
      {% block base_content%}{% endblock %}
      {% render_block "js" %}
  </body>
</html>
```

1.11.4 Database Updates

Run the following commands to upgrade your database

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

1.11.5 Static Media

Add the following to `urls.py` to serve static media when developing:

```
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
```

Also run this command to collect static files into your `STATIC_ROOT`:

```
python manage.py collectstatic
```

CHAPTER 2

Getting Started

2.1 Introductory Tutorial

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

2.1.1 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.

2.1.2 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.

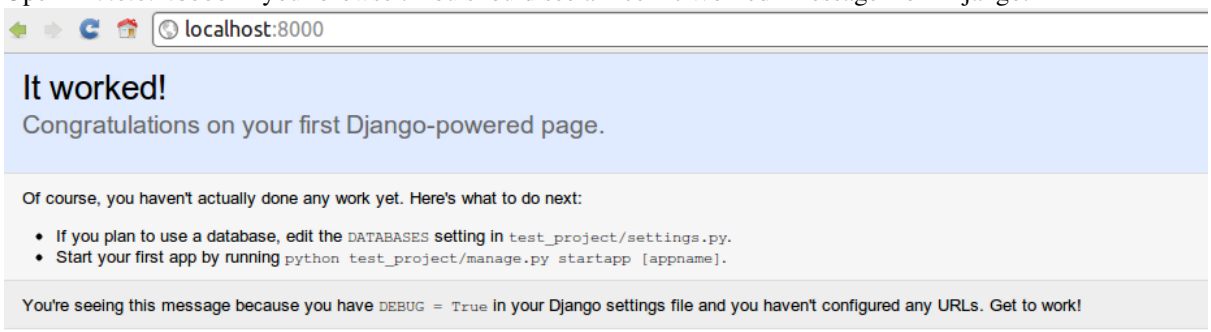
Starting your Django project

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.



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.split(os.path.abspath(os.path.dirname(__file__)))[0]
```

Add the following apps to your `INSTALLED_APPS`. This includes django CMS itself as well as its dependenices and other 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'`

Warning: Adding the `'cms.plugins.snippet'` plugin is a potential security hazard. For more information, refer to *Snippet*.

The plugins are described in more detail in chapter Plugins reference. There are even more plugins available on the django CMS extensions page.

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

you may also wish to use django-filer and its components with the django CMS plugin instead of the `cms.plugins.file`, `cms.plugins.picture`, `cms.plugins.teaser` and `cms.plugins.video` core plugins. In this case you should not add them to `INSTALLED_APPS` but add the following instead:

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

Note: See the django-filer documentation and django CMS plugin documentation for detailed installation information.

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

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

- `'reversion'`

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

```
MIDDLEWARE_CLASSES = (
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.locale.LocaleMiddleware',
    'django.middleware.doc.XViewMiddleware',
    'django.middleware.common.CommonMiddleware',
    'cms.middleware.page.CurrentPageMiddleware',
    'cms.middleware.user.CurrentUserMiddleware',
    'cms.middleware.toolbar.ToolbarMiddleware',
    'cms.middleware.language.LanguageCookieMiddleware',
)
```

You need at least the following `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 `STATIC_ROOT` to where the static files should live (that is, your images, CSS files, Javascript files, etc.):

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

For uploaded files, you will need to set up the `MEDIA_ROOT` setting:

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

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

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

```
TEMPLATE_DIRS = (
    # The docs say it should be absolute path: PROJECT_PATH is precisely one.
    # Life is wonderful!
```

```
    os.path.join(PROJECT_PATH, "templates"),
)
```

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

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

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

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

The django CMS allows you to edit all languages for which Django has built in translations. Since these are numerous, we'll limit it to English for now:

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

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

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(PROJECT_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.conf.urls.i18n import i18n_patterns
from django.contrib import admin
from django.conf import settings

admin.autodiscover()

urlpatterns = i18n_patterns('',
    url(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
```

2.1.3 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 *CMS_TEMPLATES* setting:

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

If you have followed this tutorial from the beginning, this code should already be 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>
      {% cms_toolbar %}
      {% 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 to your placeholders so you can identify them more easily 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. We highly recommended putting the `{% render_block "css" %}` tag as the 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
```

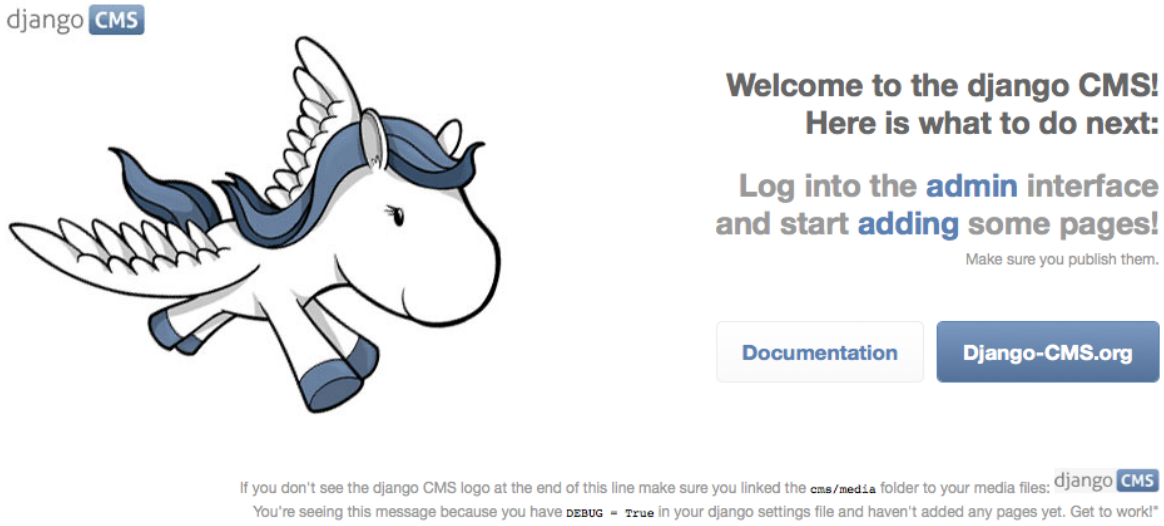
Check you did everything right

Now, use the following command to check if you did everything correctly:

```
python manage.py cms check
```

Up and running!

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



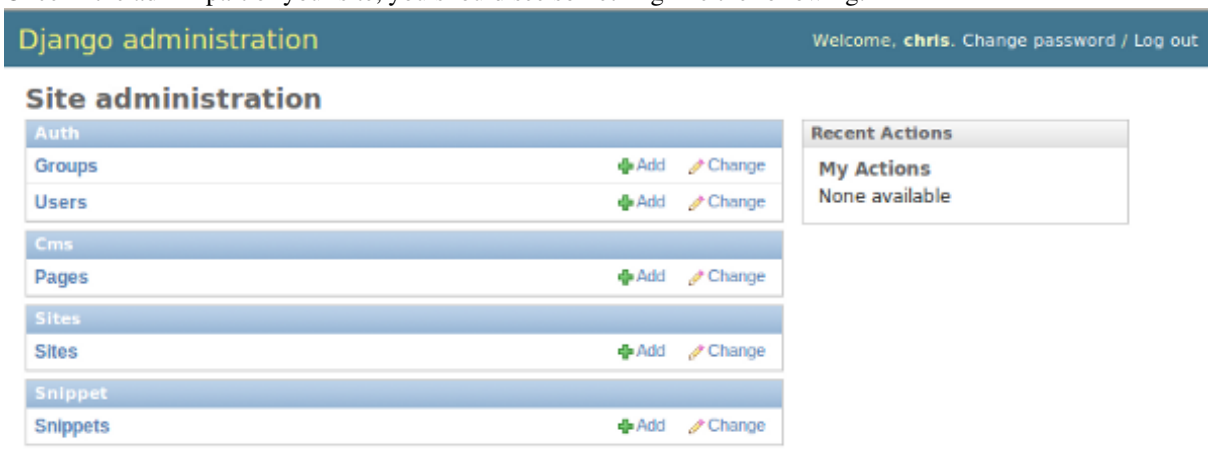
Head over to the *admin panel <http://127.0.0.1:8000/admin/>* and log in with the user you created during the database setup.

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

2.1.4 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:

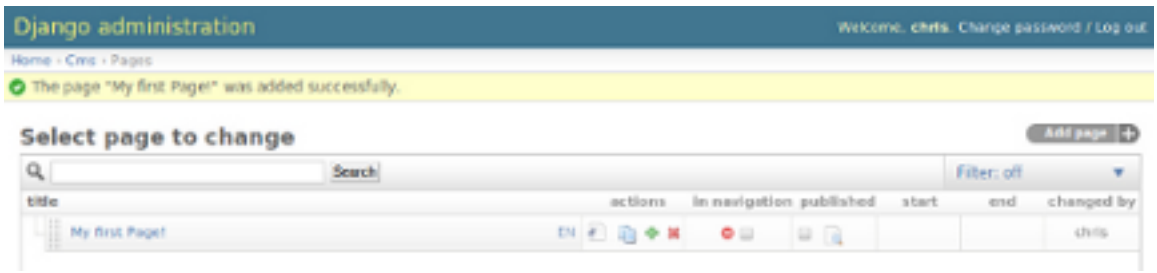


Adding a page

Adding a page is as simple as clicking "Pages" in the admin view, then the "add page" button at 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, unsurprisingly, saves the page. It will now display in the list of pages.



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

Publishing a page

The following is a list of parameters that can be changed for each of 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 at the top right-hand corner of the screen. As expected, your page is blank for the time being, since our template is a really 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:

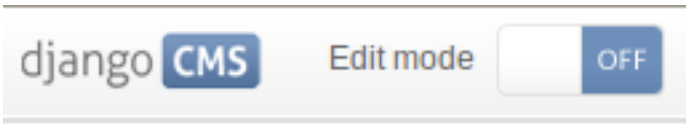
Scroll down the "Available plugins" drop-down list. This displays the plugins you added to your `INSTALLED_APPS` settings. Choose the "text" plugin in the drop-down, then press the "Add" button.

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

In the editor, type in some text and then press the "Save" button.

The new text is only visible on the draft copy so far, but you can see it by using the top button "Preview draft". If you use the "View on site" button instead, you can see that the page is still blank to the normal users.

To publish the changes you have made, click on the "Publish draft" 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 to build great websites!

2.2 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 why you should use South and how to get started quickly.

2.2.1 Installation

As always using Django and Python is a joy. Installing South is as simple as typing:

```
pip install South
```

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

2.2.2 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 will be created in the database and 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. Next run `manage.py migrate` to apply the newly created migration.

2.2.3 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.

2.3 Configuration

The django CMS has a lot of settings you can use to customize your installation so that it is exactly as you'd like it to be.

2.3.1 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 *CMS_TEMPLATES* must contain at least the `js` and `css` sekizai namespaces, for more information, see *Static files handling with sekizai*.

Warning: django CMS internally relies on a number of templates to function correctly. These exist beneath `cms` within the templates directory. As such, it is highly recommended you avoid using the same directory name for your own project templates.

2.3.2 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 granularly 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 and the 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 plugins' context before rendering. See Custom Plugins for more information.

CMS_PLUGIN_PROCESSORS

Default: `[]`

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

CMS_APPHOOKS

Default: `()`

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

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

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

Example:

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

PLACEHOLDER_FRONTEND_EDITING

Default: `True`

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

2.3.3 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 thing to do is to head over to the Wymeditor examples page in order to 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`

2.3.4 I18N and L10N

CMS_LANGUAGES

Default: Value of `LANGUAGES` converted to this format

Defines the languages available in django CMS.

Example:

```
CMS_LANGUAGES = {
    1: [
        {
            'code': 'en',
            'name': gettext('English'),
            'fallbacks': ['de', 'fr'],
            'public': True,
            'hide_untranslated': True,
            'redirect_on_fallback':False,
        },
        {
            'code': 'de',
            'name': gettext('Deutsch'),
            'fallbacks': ['en', 'fr'],
            'public': True,
        },
        {
            'code': 'fr',
            'name': gettext('French'),
            'public': False,
        },
    ],
    2: [
        {
            'code': 'nl',
            'name': gettext('Dutch'),
            'public': True,
            'fallbacks': ['en'],
        },
    ],
    'default': {
        'fallbacks': ['en', 'de', 'fr'],
        'redirect_on_fallback':True,
        'public': False,
        'hide_untranslated': False,
    }
}
```

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

CMS_LANGUAGES has different options where you can granular define how different languages behave.

On the first level you can define SITE_IDs and default values. In the example above we define two sites. The first site has 3 languages (English, German and French) and the second site has only Dutch. The *default* node defines default behavior for all languages. You can overwrite the default settings with language specific properties. For example we define *hide_untranslated* as False globally. The English language overwrites this behavior.

Every language node needs at least a *code* and a *name* property. *code* is the iso 2 code for the language. And name is the verbose name of the language.

Note: With a gettext() lambda function you can make language names translatable. To enable this add *gettext* =

lambda s: s at the beginning of your settings file. But maybe you want to leave the language name as it is.

What are the properties a language node can have?

code

String. RFC5646 code of the language.

Example: `"en"`.

Note: Is required for every language.

name

String. The verbose name of the language.

Note: Is required for every language.

public

Is this language accessible 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.

Type: Boolean Default: `True`

fallbacks

A list of languages that are used if a page is not translated yet. The ordering is relevant.

Example: `['de', 'fr']` Default: `[]`

hide_untranslated

Should untranslated pages be hidden in the menu?

Type: Boolean Default: `True`

redirect_on_fallback

If a page is not available should there be a redirect to a language that is, or should the content be displayed in the other language in this page?

Type: Boolean Default:`True`

Unicode support for automated slugs

The django CMS supports automated slug generation from page titles that contain unicode characters via the unihandecode.js project. To enable support for unihandecode.js, at least *CMS_UNIHANDECODE_HOST* and *CMS_UNIHANDECODE_VERSION* must be set.

CMS_UNIHANDECODE_HOST

default: `None`

Must be set to the URL where you host your unihandecode.js files. For licensing reasons, the django CMS does not include unihandecode.js.

If set to `None`, the default, unihandecode.js is not used.

Note: Unihandecode.js is a rather large library, especially when loading support for Japanese. It is therefore very important that you serve it from a server that supports gzip compression. Further, make sure that those files can be cached by the browser for a very long period.

CMS_UNIHANDECODE_VERSION

default: `None`

Must be set to the version number (eg `'1.0.0'`) you want to use. Together with *CMS_UNIHANDECODE_HOST* this setting is used to build the full URLs for the javascript files. URLs are built like this: `<CMS_UNIHANDECODE_HOST>-<CMS_UNIHANDECODE_VERSION>.<DECODER>.min.js`.

CMS_UNIHANDECODE_DECODERS

default: `['ja', 'zh', 'vn', 'kr', 'diacritic']`

If you add additional decoders to your `CMS_UNIHANDECODE_HOST`, you can add them to this setting.

CMS_UNIHANDECODE_DEFAULT_DECODER

default: `'diacritic'`

The default decoder to use when unihandecode.js support is enabled, but the current language does not provide a specific decoder in *CMS_UNIHANDECODE_DECODERS*. If set to `None`, failing to find a specific decoder will disable unihandecode.js for this language.

2.3.5 Media Settings

CMS_MEDIA_PATH

default: `cms/`

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

CMS_MEDIA_ROOT

Default: `MEDIA_ROOT` + *CMS_MEDIA_PATH*

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

CMS_MEDIA_URL

default: `MEDIA_URL` + *CMS_MEDIA_PATH*

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

CMS_PAGE_MEDIA_PATH

Default: `'cms_page_media/'`

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

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

2.3.6 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 to which visitors will be redirected 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_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.

2.3.7 Advanced Settings

CSRF_COOKIE_NAME

In case you've overwritten the default Django *CSRF_COOKIE_NAME* setting, then you should inform Django-CMS about this by using a context processor dedicated for this. Extend the list of *TEMPLATE_CONTEXT_PROCESSORS* with

```
'cms.context_processors.csrf_cookie_name',
```

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 to which he is allowed access.

CMS_RAW_ID_USERS

Default: `False`

This setting only applies if *CMS_PERMISSION* is `True`

The "view restrictions" and "page permissions" inlines on the `cms.models.Page` admin change forms can cause performance problems where there are many thousands of users being put into simple select boxes. If set to a positive integer, this setting forces the inlines on that page to use standard Django admin raw ID widgets rather than select boxes if the number of users in the system is greater than that number, dramatically improving performance.

Note: Using raw ID fields in combination with `limit_choices_to` causes errors due to excessively long URLs if you have many thousands of users (the PKs are all included in the URL of the popup window). For this reason, we only apply this limit if the number of users is relatively small (fewer than 500). If the number of users we need to limit to is greater than that, we use the usual input field instead unless the user is a CMS superuser, in which case we bypass the limit. Unfortunately, this means that non-superusers won't see any benefit from this setting.

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_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 *show_placeholder* and *page_url* template tags.

Note: This settings was previously called `CMS_CONTENT_CACHE_DURATION`

'menus'

Default: `3600`

Cache expiration (in seconds) for the menu tree.

Note: This settings was previously called `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

CMS_MAX_PAGE_PUBLISH_REVERSIONS

Default: 25

If django-reversion is installed everything you do with a page and all plugin changes will be saved in a revision. In the page admin there is a history button to revert to previous version of a page. In the past we had the problem with huge databases from the revision tables after some time. As a mitigation when you publish a page all revisions that are not publish revision will be deleted. This setting however declares how many publish revisions are saved in the database. By default the newest 25 publish revisions are kept and all other are deleted when you publish a page. If you set this to 0 all publish revisions are kept but you are responsible to keep the revision table small.

2.4 Navigation

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

- *show_menu*
- *show_menu_below_id*
- *show_sub_menu*
- *show_breadcrumb*

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.

2.4.1 show_menu

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

The first two parameters, `start_level` (default=0) and `end_level` (default=100) specify from which level the navigation shoud be rendered and at which level it should stop. If you have 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" %}
```

2.4.2 show_menu_below_id

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

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

You can give it the same optional parameters as *show_menu*:

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

2.4.3 show_sub_menu

Displays the sub menu of the current page (as a nested list).

The first argument, `levels` (default=100), specifies how many levels deep the sub menu should be displayed

The second argument, `root_level` (default=None), specifies at what level, if any, the menu should root at. For example, if root_level is 0 the menu will start at that level regardless of what level the current page is on.

The third argument, `nephews` (default=100), specifies how many levels of nephews (children of siblings) are show.

The template can be found at `cms/sub_menu.html`:

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

Rooted at level 0:

```
<ul>
    {% show_sub_menu 1 0 %}
</ul>
```

Or with a custom template:

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

2.4.4 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.

2.4.5 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 would still have 0 as its *menu_level*.

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

The absolute URL of the node, without any protocol, domain or port.

```
{{ node.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".

2.4.6 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.

2.4.7 Modifying & Extending the menu

Please refer to the App Integration documentation

2.5 Plugins reference

2.5.1 File

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

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

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

You should take care that the directory defined by the configuration setting *`CMS_PAGE_MEDIA_PATH`* (by default `cms_page_media/` relative to `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 only works with local storages. If you need more advanced solutions, please look at alternative file plugins for the django CMS, such as django-filer.

2.5.2 Flash

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

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

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

2.5.3 GoogleMap

Displays a map of an address on your page.

Both address and coordinates are supported to center the map; zoom level and route planner can be set when adding/editing plugin in the admin.

New in version 2.3.2: width/height parameter has been added, so it's no longer required to set plugin container size in CSS or template.

Changed in version 2.3.2: Zoom level is set via a select field which ensure only legal values are used.

Note: Due to the above change, *level* field is now marked as *NOT NULL*, and a datamigration has been introduced to modify existing googlemap plugin instance to set the default value if *level* if is *NULL*.

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

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

2.5.4 Link

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

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

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

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

2.5.5 Picture

Displays a picture in a page.

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

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

There are several solutions for Python and Django out there to automatically resize your pictures, you can find some on Django Packages and compare them there.

In your project template directory create a folder called cms/plugins and in it create a file called picture.html. Here is an example picture.html template using easy-thumbnails:

```
{% load thumbnail %}

{% if link %}<a href="{{ link }}">{% endif %}
{% if placeholder == "content" %}
    <img src="{% thumbnail picture.image 300x600 %}"{% if picture.alt %} alt="{{ picture.alt }}"{
{% else %}
    {% if placeholder == "teaser" %}
        <img src="{% thumbnail picture.image 150x150 %}"{% if picture.alt %} alt="{{ picture.alt
    {% endif %}
{% endif %}
{% if link %}</a>{% endif %}
```

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 *CMS_PAGE_MEDIA_PATH* (by default cms_page_media/ relative to MEDIA_ROOT) is writable by the user under which django will be running.

Note: In order to improve clarity, some Picture fields have been omitted in the example template code.

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_image component instead.

2.5.6 Snippet

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

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

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

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

Warning: This plugin is a potential security hazard, since it allows admins to place custom JavaScript on pages. This may allow administrators with the right to add snippets to elevate their privileges to superusers. This plugin should only be used during the initial development phase for rapid prototyping and should be disabled on production sites.

2.5.7 Teaser

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

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

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

You should take care that the directory defined by the configuration setting *CMS_PAGE_MEDIA_PATH* (by default `cms_page_media/` relative to `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.

2.5.8 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 `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 `INSTALLED_APPS` setting:

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

2.5.9 Video

Plays Video Files or Youtube / Vimeo Videos. Uses the OSFlashVideoPlayer. When uploading videos use either .flv files or h264 encoded video files.

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

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

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

- `VIDEO_AUTOPLAY` ((default: `False`)
- `VIDEO_AUTOHIDE` (default: `False`)
- `VIDEO_FULLSCREEN` (default: `True`)
- `VIDEO_LOOP` (default: `False`)
- `VIDEO_AUTOPLAY` (default: `False`)
- `VIDEO_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 *`CMS_PAGE_MEDIA_PATH`* (by default `cms_page_media/` relative to `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.

2.5.10 Twitter

Display's a number of a twitter user's latest posts.

For installation be sure you have the following in your project's `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.

2.5.11 Inherit

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

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

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

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

2.6 Common issues

2.6.1 Caught MultipleObjectsReturned while rendering

After upgrading to a new version with an existing database, you encounter something like:

Caught MultipleObjectsReturned while rendering: get() returned more than one CacheKey – it returned 12! Lookup parameters were {'key': 'cms-menu_nodes_en_1_1_user', 'language': 'en', 'site': 1L}

What has happened is that your database contains some old cache data in the *menus_cachekey* table. Just delete all those entries.

CHAPTER 3

Advanced

3.1 Internationalization

3.1.1 Multilingual URLs

If you use more than one language, django CMS urls need to be referenced via `i18n_patterns()`. For more information about this see the official django documentation.

Main *urls.py* example:

```
from django.conf import settings
from django.conf.urls.defaults import patterns, include, url
from django.contrib import admin
from django.conf.urls.i18n import i18n_patterns
from django.contrib.staticfiles.urls import staticfiles_urlpatterns

admin.autodiscover()

urlpatterns = patterns('',
    url(r'^jsi18n/(?P<packages>\S+?)/$', 'django.views.i18n.javascript_catalog'),
)

urlpatterns += staticfiles_urlpatterns()

urlpatterns += i18n_patterns('',
    url(r'^admin/', include(admin.site.urls)),
    url(r'^', include('cms.urls')), # <--------- include the django cms urls via i18n_patterns
)
```

3.1.2 Language Cookie

By default if someone visits a page at *http://www.mysite.fr/* django determines the language as follow:

- language in url
- language in session
- language in cookie
- language in from browser
- LANGUAGE_CODE from settings

No if i have a German browser and visit a page that is only English and French, it will choose the language that is in LANGUAGE_CODE. If this is English but i only speak French i have to switch the language. No if after sometime i visit the page again. The page will display in English again and I have to switch again. The same is for every link that points to / will switch to English again. To fix this behavior the cms ships with a middleware:

cms.middleware.language.LanguageCookieMiddleware

add this to you middleware settings fix this.

3.1.3 Language Chooser

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

Example:

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

3.1.4 page_language_url

This template tag returns the URL of the current page in another language.

Example:

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

3.1.5 hide_unstranslated

If you add a default directive to your *CMS_LANGUAGES* with a `hide_unstranslated` to `False` all pages will be displayed in all languages even if they are not translated yet.

If *hide_untranslated* is `True` in your *CMS_LANGUAGES* and you are on a page that doesn't yet have an English translation 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.

3.1.6 Automated slug generation unicode characters

If your site has languages which use non-ASCII character sets, you might want to enable *CMS_UNIHANDECODE_HOST* and *CMS_UNIHANDECODE_VERSION* to get automated slugs for those languages too.

3.2 Sitemap Guide

3.2.1 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

3.2.2 Configuration

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

3.2.3 django.contrib.sitemaps

More information about `django.contrib.sitemaps` can be found in the official Django documentation.

3.3 Template Tags

3.3.1 CMS templatetags

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

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

placeholder

Changed in version 2.1: The placeholder name became case sensitive.

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 in the event that the placeholder has no plugins or the plugins do not generate any output.

Example:

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

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

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

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

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

This will walk up the page tree up until 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 *CMS_PLACEHOLDER_CONF* setting where you can also add extra context variables and change some other placeholder behavior.

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 to 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 %}
```

show_uncached_placeholder

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

Arguments:

- `placeholder_name`

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

Example:

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

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>
```

If a matching page isn't found and `DEBUG` is `True`, an exception will be raised. However, if `DEBUG` is `False`, an exception will not be raised. Additionally, if `SEND_BROKEN_LINK_EMAILS` is `True` and you have specified some addresses in `MANAGERS`, an email will be sent to those addresses to inform them of the broken link.

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 %}
```

New in version 2.3.2: This template tag supports the `as` argument. With this you can assign the result of the template tag to a new variable that you can use elsewhere in the template.

Example:

```
{% page_attribute "page_title" as title %}
<title>{{ title }}</title>
```

It even can be used in combination with the `page_lookup` argument.

Example:

```
{% page_attribute "page_title" "my_page_reverse_id" as title %}
<a href="/mypage/">{{ title }}</a>
```

New in version 2.4.

render_plugin

This templatetag is used to render child plugins of the current plugin and should be used inside plugin templates.

Arguments:

- `plugin`

Plugin needs to be an instance of a plugin model.

Example:

```
{% load cms_tags %}
<div class="multicolumn">
{% for plugin in instance.child_plugins %}
        <div style="width: {{ plugin.width }}00px;">
        {% render_plugin plugin %}
        </div>
{% endfor %}
</div>
```

Normally the children of plugins can be accessed via the `child_plugins` atrribute of plugins. Plugins need the `allow_children` attribute to set to *True* for this to be enabled.

3.3.2 Menu Templatetags

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

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

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 which level the navigation shoud be rendered and at which level it should stop. If you have 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.

show_menu 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" %}
```

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>
```

show_sub_menu

Displays the sub menu of the current page (as a nested list).

The first argument, `levels` (default=100), specifies how many levels deep the submenu should be displayed

The second argument, `root_level` (default=None), specifies at what level, if any, the menu should root at. For example, if root_level is 0 the menu will start at that level regardless of what level the current page is on.

The third argument, `nephews` (default=100), specifies how many levels of nephews (children of siblings) are show.

The template can be found at `cms/sub_menu.html`:

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

Rooted at level 0:

```
<ul>
    {% show_sub_menu 1 0 %}
</ul>
```

Or with a custom template:

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

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 %}
```

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.

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 the last argument to the `language_chooser tag` as a string. In "raw" mode, the language will be displayed like its 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 menus.utils.

For more information, see Internationalization.

3.3.3 Toolbar Templatetags

The `cms_toolbar` templatetag is included in the `cms_tags` library and will add the required css and javascript to the sekizai blocks in the base template. The templatetag has to be placed after the `<body>` tag and before any `{% cms_placeholder %}` occurrences within your HTML.

Example:

```
<body>
{% cms_toolbar %}
{% placeholder "home" %}
...
```

3.4 Command Line Interface

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

```
python manage.py cms
```

3.4.1 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.

`cms list plugins` will issue warnings when it finds orphaned plugins (see `cms delete_orphaned_plugins` below).

cms check

Checks your configuration and environment.

3.4.2 Plugin and apphook management commands

cms delete_orphaned_plugins

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

Identifies and deletes orphaned plugins.

Orphaned plugins are ones that exist in the CMSPlugins table, but:

- have a plugin_type that is no longer even installed
- have no corresponding saved instance in that particular plugin type's table

Such plugins will cause problems when trying to use operations that need to copy pages (and thefore plugins), which includes `cms moderator on` as well as page copy operations in the admin.

It is advised to run `cms list plugins` periodically, and `cms delete_orphaned_plugins` when required.

cms uninstall

The `uninstall` subcommand can be used to make uninstalling 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 in 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 in the django CMS. If you are unsure about the apphook name, use the *cms list* to see a list of installed apphooks.

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

3.4.3 Moderation commands

`cms moderator`

If you migrate from an earlier version, you should use the `cms moderator on` command to ensure that your published pages are up to date, whether or not you used moderation in the past.

> **Warning:** This command **alters data** in your database. You should make a backup of your database before using it! **Never** run this command without first checking for orphaned plugins, using the `cms list plugins` command, and if necessary `delete_orphaned_plugins`. Running `cms moderator` with orphaned plugins will fail and leave bad data in your database.

3.4.4 MPTT repair command

`cms mptt-repair`

Occasionally, the MPTT structure in which pages and plugins are held can accumulate small errors. These are typically the result of failed operations or large and complex restructurings of the tree (perhaps even cosmic rays, planetary alignment or other mysterious conditions).

Usually you won't even notice them, and they won't affect the operation of the system, but when you run into trouble it's useful to be able to rebuild the tree - it's also useful to rebuild it as part of preventative maintenance.

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

3.5 Permissions

In django-cms you can set two types of permissions:

1. View restrictions for restricting front-end view access to users
2. Page permissions for allowing staff users to only have rights on certain sections of certain sites

To enable these features, `settings.py` requires:

    CMS_PERMISSION = True

3.5.1 View restrictions

View restrictions can be set-up from the *View restrictions* formset on any cms page. Once a page has at least one view restriction installed, only users with granted access will be able to see that page. Mind that this restriction is for viewing the page as an end-user (front-end view), not viewing the page in the admin interface!

View restrictions are also controlled by the `CMS_PUBLIC_FOR` setting. Possible values are `all` and `staff`. This setting decides if pages without any view restrictions are:

- viewable by everyone – including anonymous users (*all*)
- viewable by staff users only (*staff*)

3.5.2 Page permissions

After setting `CMS_PERMISSION = True` you will have three new models in the admin index:

1. Users (page)

2. User groups (page)
3. Pages global permissions

Using *Users (page)* you can easily add users with permissions over cms pages.

You would be able to create an user with the same set of permissions using the usual *Auth.User* model, but using *Users (page)* is more convenient.

A new user created using *Users (page)* with given page add/edit/delete rights will not be able to make any changes to pages straight away. The user must first be assigned to a set of pages over which he may exercise these rights. This is done using the *Page permissions* formset on any page or by using *Pages global Permissions*.

The *Page permission* formset has multiple checkboxes defining different permissions: can edit, can add, can delete, can change advanced settings, can publish, can move and can change permission. These define what kind of actions the user/group can do on the pages on which the permissions are being granted through the *Grant on* drop-down.

Can change permission refers to whether the user can change the permissions of his subordinate users. Bob is the subordinate of Alice if one of:

- Bob was created by Alice
- Bob has at least one page permission set on one of the pages on which Alice has the *Can change permissions* right

Note: Mind that even though a new user has permissions to change a page, that doesn't give him permissions to add a plugin within that page. In order to be able to add/change/delete plugins on any page, you will need to go through the usual *Auth.User* model and give the new user permissions to each plugin you want him to have access to. Example: if you want the new user to be able to use the text plugin, you will need to give him the following rights: `text | text | Can add text`, `text | text | Can change text`, `text | text | Can delete text`.

Using the *Pages global permissions* model you can give a set of permissions to all pages in a set of sites.

CHAPTER 4

Extending the CMS

4.1 Extending the CMS: Examples

From this point onwards, this tutorial assumes you have done the Django Tutorial and will show you how to integrate the tutorial's poll app into the django CMS. Hereafter, if a poll app is mentioned, we are referring to the one you get after completing the Django Tutorial. Also, make sure the poll app is in your `INSTALLED_APPS`.

We assume your main `urls.py` looks something 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')),
)
```

4.1.1 My First Plugin

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

The Model

For our polling app we would like to have a small poll plugin which shows a poll and lets the user vote.

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

```
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. After having followed the Django Tutorial and adding this file your polls app folder should look like this:

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

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

For our poll plugin, write the 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 in the above plugin class. In order for our plugin to work, that template must exist and is responsible for rendering the plugin.

The template should look something like this:

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

<form action="{% url polls.views.vote instance.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.

4.1.2 My First App (apphook)

Right now, external apps are statically hooked into the main `urls.py`. This is not the preferred approach 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 it 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
```

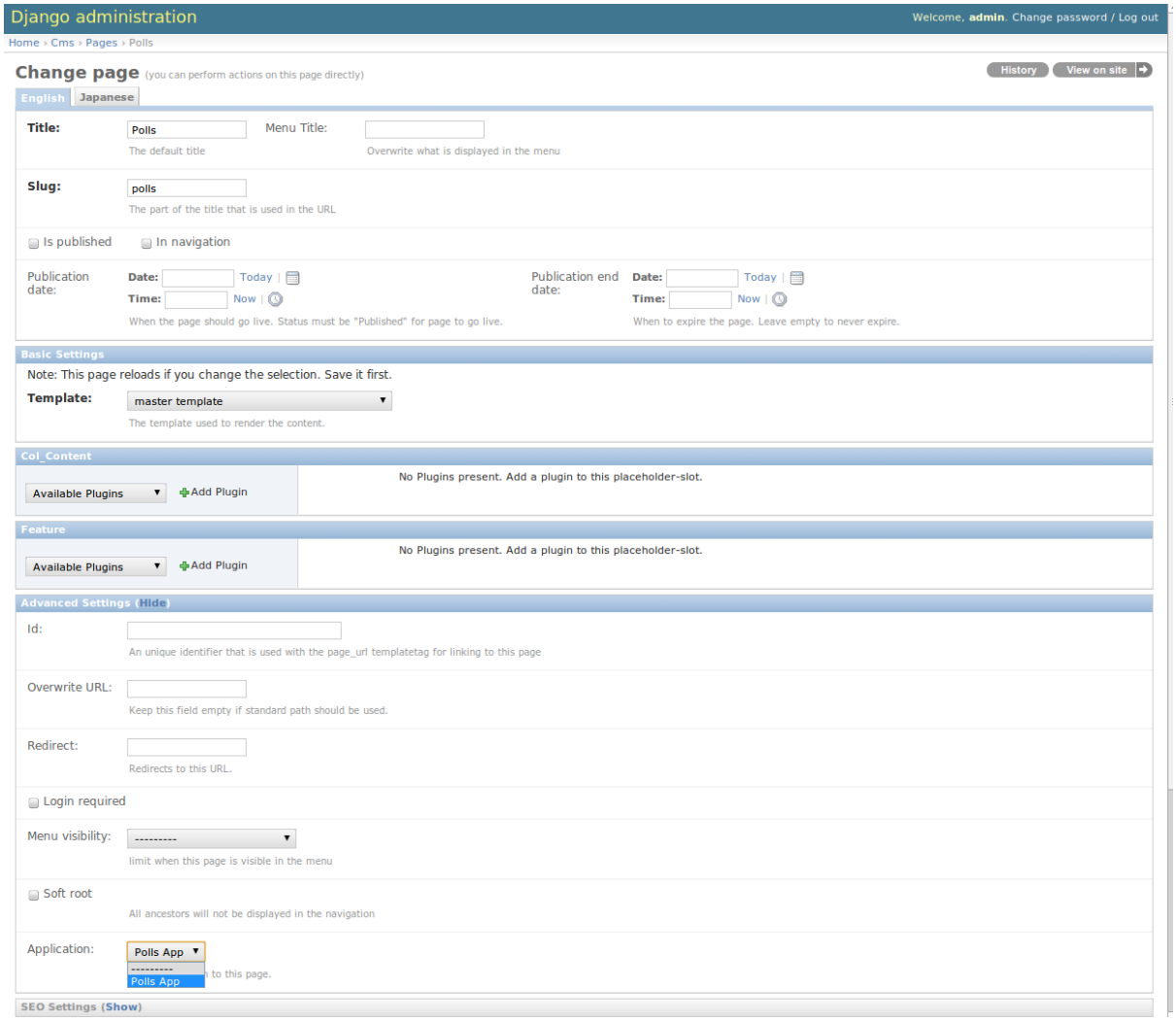
Now remove the inclusion of the polls urls in your main `urls.py` so it looks like this:

```
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'.



Unfortunately, for these changes to take effect, you will have to restart your server. So do that and afterwards if you navigate to that CMS Page, you will see your polls application.

4.1.3 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 ensure 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 its first argument, a URL as
            # its second argument and a (for this tree) unique id as its 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.
```

At this point this menu alone doesn't do a whole lot. 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)
```

4.2 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). They enable the publishing of information automatically, without further intervention.

This means that your published web content, whatever it is, is kept 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 - writing a CMS Plugin is rather simple.

4.2.1 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 a "Latest releases" box on your site's home page.

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.

4.2.2 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 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 from:

- 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 doesn't need to be configured, because it only ever does one thing.

For example, you could have a plugin that 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 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 this will require a model.

4.2.3 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. For our example, include 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
    render_template = "hello_plugin.html"

plugin_pool.register_plugin(HelloPlugin)
```

Now we're almost done. All that's left is to add 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
```

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 for storing information about this plugin. If you do not require any special information, for example configuration, to be stored for your plugins, you can simply use `cms.models.pluginmodel.CMSPlugin` (we'll look at that model more closely in a bit). In a normal admin class, you don't need to supply this information because `admin.site.register(Model, Admin)` takes care of it, but a plugin is not registered in that way.
- `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. By default the name is a nicer version of the class name.
- `render_template`: The template to render this plugin with.

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

The *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.

New in version 2.4.

By default this method will add `instance` and `placeholder` to the context, which means for simple plugins, there is no need to overwrite this method.

4.2.4 Troubleshooting

Since plugin modules are found and loaded by django's importlib, you might experience errors because the path environment is different at runtime. If your *cms_plugins* isn't loaded or accessible, try the following:

```
$ python manage.py shell
>>> from django.utils.importlib import import_module
>>> m = import_module("myapp.cms_plugins")
>>> m.some_test_function()
```

4.2.5 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 its fallback name for non-authenticated users configurable.

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

```
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
```

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 (to it, or from it) or many-to-many relations you are responsible for copying those related objects, if required, whenever the CMS copies the plugin - **it won't do it for you automatically**.

Every plugin model inherits the empty `cms.models.pluginmodel.CMSPlugin.copy_relations()` method from the base class, and it's called when your plugin is copied. So, it's there for you to adapt to your purposes as required.

Typically, you will want it to copy related objects. To do this you should create a method called `copy_relations` on your plugin model, that receives the **old** instance of the plugin as an argument.

You may however decide that the related objects shouldn't be copied - you may want to leave them alone, for example. Or, you might even want to choose some altogether different relations for it, or to create new ones when it's copied... it depends on your plugin and the way you want it to work.

If you do want to copy related objects, you'll need to do this in two slightly different ways, depending on whether your plugin has relations *to* or *from* other objects that need to be copied too:

For foreign key relations *from* other objects

Your plugin may have items with foreign keys to it, which will typically be the case if you set it up so that they are inlines in its admin. So you might have a two models, one for the plugin and one for those items:

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

class AssociatedItem(models.Model):
    plugin = models.ForeignKey(
        ArticlePluginModel,
        related_name="associated_item"
        )
```

You'll then need the `copy_relations()` method on your plugin model to loop over the associated items and copy them, giving the copies foreign keys to the new plugin:

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

    def copy_relations(self, oldinstance):
        for associated_item in oldinstance.associated_item.all():
            # instance.pk = None; instance.pk.save() is the slightly odd but
            # standard Django way of copying a saved model instance
            associated_item.pk = None
```

```
            associated_item.plugin = self
            associated_item.save()
```

For many-to-many or foreign key relations *to* other objects

Let's assume these are the relevant bits of your plugin:

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

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

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

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

If your plugins have relational fields of both kinds, you may of course need to use *both* the copying techniques described above.

4.2.6 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.

The template that the plugin editing mechanism uses is `cms/templates/admin/cms/page/plugin_change_form.html` You might need to change this.

If you want to customise this the best way to do it is:

- create a template of your own that extends `cms/templates/admin/cms/page/plugin_change_form.html` to provide the functionality you require
- provide your *cms.plugin_base.CMSPluginBase* subclass with a `change_form_template` attribute pointing at your new template

Extending `admin/cms/page/plugin_change_form.html` ensures that you'll keep a unified look and functionality across your plugins.

There are various reasons *why* you might want to do this. For example, you might have a snippet of JavaScript that needs to refer to a template variable), which you'd likely place in `{% block extrahead %}`, after a `{{ block.super }}` to inherit the existing items that were in the parent template.

Or: `cms/templates/admin/cms/page/plugin_change_form.html` extends Django's own `admin/base_site.html`, which loads a rather elderly version of jQuery, and your plugin admin might require something newer. In this case, in your custom `change_form_template` you could do something like:

```
{% block jquery %}
    <script type="text/javascript" src="///ajax.googleapis.com/ajax/libs/jquery/1.8.0/jquery.min.
{% endblock jquery %}``
```

to override the `{% block jquery %}`.

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.

Note that sekizai *can't* help you with the *admin-side* plugin templates - what follows is for your plugins' *output* templates.

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 be followed (but don't necessarily need to be):

- 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">
{% addtoblock "js" %}<script type="text/javascript" src="{{ MEDIA_URL }}myplugin/js/myotherfile.j
{% addtoblock "css" %}<link rel="stylesheet" type="text/css" href="{{ MEDIA_URL }}myplugin/css/as
{% 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">
<script type="text/javascript" src="{{ MEDIA_URL }}myplugin/js/myotherfile.js"></script>{% endaddt
{% addtoblock "css" %}
    <link rel="stylesheet" type="text/css" href="{{ MEDIA_URL }}myplugin/css/astylesheet.css"></s
{% 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 *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 *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 wrap each plugin in the main placeholder in a colored box but it would be too complicated to edit each individual plugin's template:

In your `settings.py`:

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

In your `yourapp.cms_plugin_processors.py`:

```
def wrap_in_colored_box(instance, placeholder, rendered_content, original_context):
    '''
    This plugin processor wraps each plugin's output in a colored box if it is in the "main" placeholder
    '''
    # 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
        # 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')
            'border_color': instance.border_color if hasattr(instance, 'border_color') else 'light
        })
        # Finally, render the content through that template, and return the output
        return t.render(c)
```

Plugin Attribute Reference

A list of all attributes a plugin has and that can be overwritten:

change_form_template

The template used to render the form when you edit the plugin.

Default:

admin/cms/page/plugin_change_form.html

Example:

```
class MyPlugin(CMSPluginBase):
    model = MyModel
    name = _("My Plugin")
    render_template = "cms/plugins/my_plugin.html"
        change_form_template = "admin/cms/page/plugin_change_form.html"
```

frontend_edit_template

The template used for wrapping the plugin in frontend editing.

Default:

cms/toolbar/placeholder_wrapper.html

admin_preview

Should the plugin be previewed in admin when you click on the plugin or save it?

Default: False

render_template

The path to the template used to render the template. Is required.

render_plugin

Should the plugin be rendered at all, or doesn't it have any output?

Default: True

model

The Model of the Plugin. Required.

text_enabled

Default: False Can the plugin be inserted inside the text plugin?

If this is enabled the following function need to be overwritten as well:

icon_src()

Should return the path to an icon displayed in the text.

icon_alt()

Should return the alt text for the icon.

page_only

Default: False

Can this plugin only be attached to a placeholder that is attached to a page? Set this to true if you always need a page for this plugin.

allow_children

Default: False

Can this plugin have child plugins? Or can other plugins be placed inside this plugin?

child_classes

Default: None A List of Plugin Class Names. If this is set, only plugins listed here can be added to this plugin.

4.3 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

   Statically extend the menu entries

2. Attach Menus

   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

4.3.1 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 the menu entry should read as

- 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 supply the the id of the parent here.

- 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.

4.3.2 Attach Menus

Classes that extend from `menus.base.Menu` always get attached to the root. But if you want the menu to 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`

4.3.3 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 it 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 as well `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.

Apphook Menus

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.

Application and instance namespaces

If you'd like to use application namespaces to reverse the URLs related to your app, you can assign a value to the *app_name* attribute of your app hook like this:

```
class MyNamespacedApphook(CMSApp):
    name = _("My Namespaced Apphook")
    urls = ["myapp.urls"]
    app_name = "myapp_namespace"

apphook_pool.register(MyNamespacedApphook)
```

As seen for Language Namespaces, you can reverse namespaced apps similarly:

```
{% url myapp_namespace:app_main %}
```

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

```
{% load i18n %}
{% language "de" %}
    {% url myapp_namespace:app_main %}
{% endlanguage %}
```

What makes namespaced app hooks really interesting is the fact that you can hook them up to more than one page and reverse their URLs by using their instance namespace. Django CMS takes the value of the *reverse_id* field assigned to a page and uses it as instance namespace for the app hook.

To reverse the URLs you now have two different ways: explicitly by defining the instance namespace, or implicitely by specifiyng the application namespace and letting the *url* templatetag resolving the correct application instance by looking at the currently set *current_app* value.

Note: The official Django documentation has more details about application and instance namespaces, the *current_app* scope and the reversing of such URLs. You can look it up at https://docs.djangoproject.com/en/dev/topics/http/urls/#url-namespaces

---

When using the *reverse* function, the *current_app* has to be explicitly passed as an argument. You can do so by looking up the *current_app* attribute of the request instance:

```
def myviews(request):
    ...
    reversed_url = reverse('myapp_namespace:app_main',
            current_app=request.current_app)
    ...
```

Or, if you are rendering a plugin, of the context instance:

```
class MyPlugin(CMSPluginBase):
    def render(self, context, instance, placeholder):
        ...
        reversed_url = reverse('myapp_namespace:app_main',
                current_app=context.current_app)
        ...
```

4.3.4 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. 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)
```

4.3.5 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.

4.4 API References

4.4.1 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 module does 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 *CMS_TEMPLATES*
- **language** (*string*) – Language code for this page. Must be in 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 *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 is creating 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 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 *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'*, *target=None*, ***data*)

Adds a plugin to a placeholder and returns it.

Parameters

- **placeholder** (cms.models.placeholdermodel.Placeholder in-

stance) – 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 LANGUAGES
- **position** (*string*) – Position to add this plugin to the placeholder, must be a valid django-mptt position
- **target** – Parent plugin. Must be plugin instance
- **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*, *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.permissionmodels.ACCESS_PAGE, cms.models.permissionmodels.ACCESS_CHILDREN, cms.models.permissionmodels.ACCESS_DESCENDANTS or cms.models.permissionmodels.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')
```

4.4.2 cms.constants

cms.constants.**TEMPLATE_INHERITANCE_MAGIC**
The token used to identify when a user selects "inherit" as template for a page.

4.4.3 cms.plugin_base

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

admin_preview
Defaults to False, if True there will be a preview 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 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

4.4.4 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.

4.5 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.

4.5.1 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 its 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.

Warning: For security reasons the related_name for a `PlaceholderField` may not be surpressed using `'+'` to allow the cms to check permissions properly. Attempting to do so will raise a `ValueError`.

Admin Integration

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.models import MyModel

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

I18N Placeholders

Out of the box `PlaceholderAdmin` supports multiple languages and will display language tabs. If you extend *PlaceholderAdmin* and overwrite *change_form_template* be sure to have a look at 'admin/placeholders/placeholder/change_form.html' on how to display the language tabs.

If you need other fields then the placeholders translated as well: django CMS has support for django-hvad. If you use a *TranslatableModel* model be sure to not include the placeholder fields in the translated fields:

```
class MultilingualExample1(TranslatableModel):
    translations = TranslatedFields(
        title=models.CharField('title', max_length=255),
        description=models.CharField('description', max_length=255),
    )
    placeholder_1 = PlaceholderField('placeholder_1')

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

Be sure to combine both hvad's `TranslatableAdmin` and `PlaceholderAdmin` when registering your model with the admin site:

```
from cms.admin.placeholderadmin import PlaceholderAdmin
from django.contrib import admin
from hvad.admin import TranslatableAdmin
from myapp.models import MultilingualExample1

class MultilingualModelAdmin(TranslatableAdmin, PlaceholderAdmin):
    pass

admin.site.register(MultilingualExample1, MultilingualModelAdmin)
```

Templates

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

```
{% load placeholder_tags %}

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

The `render_placeholder` tag takes a `PlaceholderField` instance as its first argument and optionally accepts a width parameter as its second argument for context sensitive plugins. The view in which you render your placeholder field must return the `request` object in the context. This is typically achieved in Django applications by using `RequestContext`:

```
from django.shortcuts import get_object_or_404, render_to_response
from django.template.context import RequestContext
from myapp.models import MyModel

def my_model_detail(request, id):
    object = get_object_or_404(MyModel, id=id)
    return render_to_response('my_model_detail.html', {
        'object': object,
    }, context_instance=RequestContext(request))
```

If you want to render plugins from a specific language, you can use the tag like this:

```
{% load placeholder_tags %}

{% render_placeholder mymodel_instance.my_placeholder language 'en' %}
```

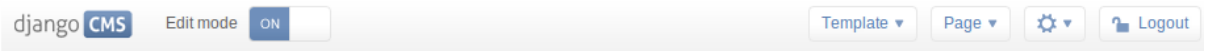
4.5.2 Adding content to a placeholder

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

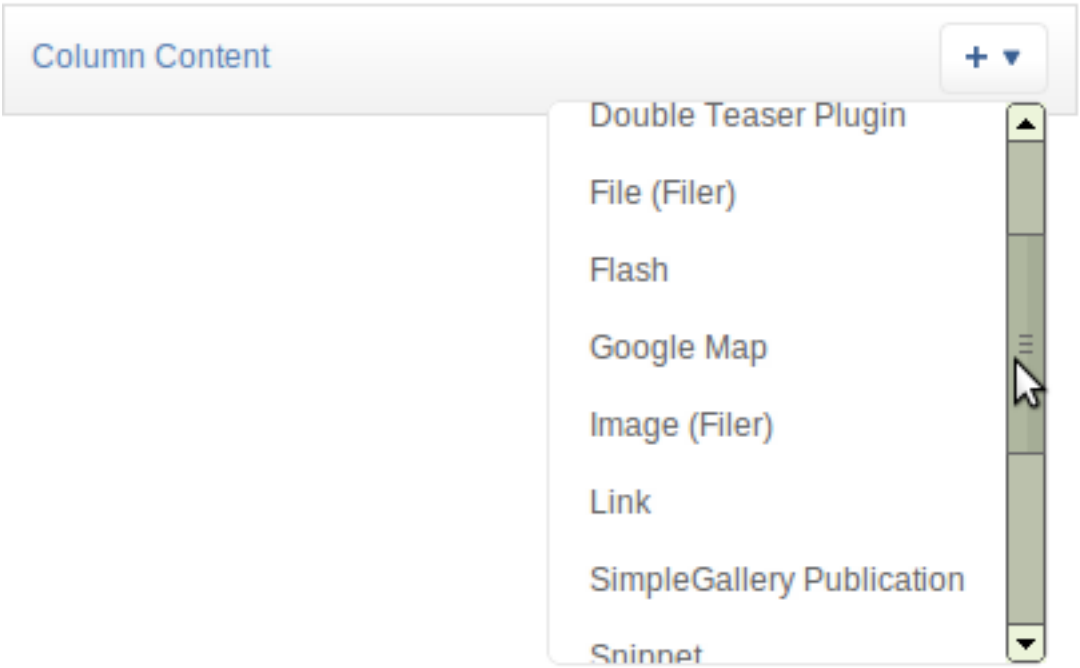
Using the front-end editor

Probably the simplest way to add content to a placeholder, simply visit the page displaying your model (where you put the `render_placeholder` tag), then append `?edit` to the page's URL. This will make a top banner appear, and after switching the "Edit mode" button to "on", the banner will prompt you for your username 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 get an idea how it will look *in fine*. However, to view the final look of a plugin simply leave edit mode by clicking the "Edit mode" button in the banner again.

4.5.3 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 its own `fieldset`, one `PlaceholderField` per fieldset.
2. You **must** include the following two classes: `'plugin-holder'` and `'plugin-holder-nopage'`

4.6 Search and the django CMS

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

4.7 Form and model fields

4.7.1 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.

4.7.2 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`.

4.8 Testing Your Extenstions

4.8.1 Testing Apps

Resolving View Names

Your apps need testing, but in your live site they aren't in `urls.py` as they are attached to a CMS page. So if you want to be able to use `reverse()` in your tests, or test templates that use the `url` template tag, you need to hook up your app to a special test version of `urls.py` and tell your tests to use that.

So you could create `myapp/tests/test_urls.py` with the following code:

```
from django.contrib import admin
from django.conf.urls import url, patterns, include

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

And then in your tests you can plug this in with the `override_settings()` decorator:

```
from django.test.utils import override_settings
from cms.test_utils.testcases import CMSTestCase

class MyappTests(CMSTestCase):

    @override_settings(ROOT_URLCONF='myapp.tests.test_urls')
    def test_myapp_page(self):
        test_url = reverse('myapp_view_name')
        # rest of test as normal
```

If you want to the test url conf throughout your test class, then you can use apply the decorator to the whole class:

```
from django.test.utils import override_settings
from cms.test_utils.testcases import CMSTestCase

@override_settings(ROOT_URLCONF='myapp.tests.test_urls')
class MyappTests(CMSTestCase):

    def test_myapp_page(self):
        test_url = reverse('myapp_view_name')
        # rest of test as normal
```

CMSTestCase

Django CMS includes `CMSTestCase` which has various utility methods that might be useful for testing your CMS app and manipulating CMS pages.

4.8.2 Testing Plugins

Plugins can just be created as objects and then have methods called on them. So you could do:

```
from django.test import TestCase
from myapp.cms_plugins import MyPlugin
from myapp.models import MyappPlugin as MyappPluginModel

class MypluginTests(TestCase):

    def setUp(self):
        self.plugin = MyPlugin()

    def test_plugin(self):
        context = {'info': 'value'}
        instance = MyappPluginModel(num_items=3)
        rendered_html = self.plugin.render(context, instance, None)
        self.assertIn('string', rendered_html)
```

Sometimes you might want to add a placeholder - say to check how the plugin renders when it is in different size placeholders. In that case you can create the placeholder directly and pass it in:

```
from django.test import TestCase
from cms.api import add_plugin
from myapp.cms_plugins import MyPlugin
from myapp.models import MyappPlugin as MyappPluginModel

class ImageSetTypePluginMixinContainerWidthTests(TestCase):
    def setUp(self):
        self.placeholder = Placeholder(slot=u"some_slot")
        self.placeholder.save()
        self.plugin = add_plugin(
            self.placeholder,
            u"MyPlugin",
            u"en",
            num_items=3,
        )
```

CHAPTER 5

Concepts

5.1 Introduction

This section doesn't explain how to do anything, but explains and analyses some key concepts in django CMS.

5.2 How the menu system works

5.2.1 Basic concepts

Registration

The menu system isn't monolithic. Rather, it is composed of numerous active parts, many of which can operate independently of each other.

What they operate on is a list of menu nodes, that gets passed around the menu system, until it emerges at the other end.

The main active parts of the menu system are menu *generators* and *modifiers*.

Some of these parts are supplied with the menus application. Some come from other applications (from the cms application in django CMS, for example, or some other application entirely).

All these active parts need to be registered within the menu system.

Then, when the time comes to build a menu, the system will ask all the registered menu generators and modifiers to get to work on it.

Generators and Modifiers

Menu generators and modifiers are classes.

Generators

To add nodes to a menu a generator is required.

There is one in cms for example, which examines the Pages in the database and adds them as nodes.

These classses are subclasses of `menus.base.Menu`. The one in cms is `cms.menu.CMSMenu`.

In order to use a generator, its `get_nodes()` method must be called.

Modifiers

A modifier examines the nodes that have been assembled, and modifies them according to its requirements (adding or removing them, or manipulating their attributes, as it sees fit).

An important one in cms (`cms.menu.SoftRootCutter`) removes the nodes that are no longer required when a soft root is encountered.

These classes are subclasses of `menus.base.Modifier`. Examples are `cms.menu.NavExtender` and `cms.menu.SoftRootCutter`.

In order to use a modifier, its `modify()` method must be called.

Note that each Modifier's `modify()` method can be called *twice*, before and after the menu has been trimmed.

For example when using the {% show_menu %} templatetag, it's called:

- first, by `menus.menu_pool.MenuPool.get_nodes()`, with the argument post_cut = False
- later, by the templatetag, with the argument post_cut = True

This corresponds to the state of the nodes list before and after `menus.templatetags.menu_tags.cut_levels()`, which removes nodes from the menu according to the arguments provided by the templatetag.

This is because some modification might be required on *all* nodes, and some might only be required on the subset of nodes left after cutting.

Nodes

Nodes are assembled in a tree. Each node is an instance of the *`menus.base.NavigationNode`* class.

A NavigationNode has attributes such as URL, title, parent and children - as one would expect in a navigation tree.

> **Warning:** You can't assume that a *`menus.base.NavigationNode`* represents a django CMS Page. Firstly, some nodes may represent objects from other applications. Secondly, you can't expect to be be able to access Page objects via NavigationNodes.

5.2.2 How does all this work?

Tracing the logic of the menu system

Let's look at an example using the {% show_menu %} templatetag. It will be different for other templatetags, and your applications might have their own menu classes. But this should help explain what's going on and what the menu system is doing.

One thing to understand is that the system passes around a list of `nodes`, doing various things to it.

Many of the methods below pass this list of nodes to the ones it calls, and return them to the ones that they were in turn called by.

Don't forget that show_menu recurses - so it will do *all* of the below for *each level* in the menu.

- **`{% show_menu %}` - the templatetag in the template**
  - **`menus.templatetags.menu_tags.ShowMenu.get_context()`**
    - **`menus.menu_pool.MenuPool.get_nodes()`**
      - **`menus.menu_pool.MenuPool.discover_menus()` checks every application**

        Menu classes, placing them in the self.menus dict

        Modifier classes, placing them in the self.modifiers list

- **menus.menu_pool.MenuPool._build_nodes()**

  checks the cache to see if it should return cached nodes

  loops over the Menus in self.menus (note: by default the only generator is cms.me

  call its `get_nodes()` - the menu generator

  `menus.menu_pool._build_nodes_inner_for_one_menu()`

  adds all nodes into a big list

- **menus.menu_pool.MenuPool.apply_modifiers()**

  `menus.menu_pool.MenuPool._mark_selected()`

  loops over each node, comparing its URL with the request.path, and marks the best match as `selected`

  loops over the Modifiers in self.modifiers calling each one's modify(post_cut=

  `cms.menu.NavExtender`

  `cms.menu.SoftRootCutter` removes all nodes below the appropriate soft root

  `menus.modifiers.Marker` loops over all nodes; finds selected, marks its ancestors, siblings and children

  `menus.modifiers.AuthVisibility` removes nodes that require authorisation to see

  menus.modifiers.Level loops over all nodes; for each one that is a root

  `menus.modifiers.Level.mark_levels()` recurses over a node's descendants marking their levels

* we're now back in `menus.templatetags.menu_tags.ShowMenu.get_context()` again
* if we have been provided a root_id, get rid of any nodes other than its descendants
* `menus.templatetags.menu_tags.cut_levels()` removes nodes from the menu according to the arguments provided by the templatetag
* **menu_pool.MenuPool.apply_modifiers(post_cut = True)() loops over all the M**

  - `cms.menu.NavExtender`
  - `cms.menu.SoftRootCutter`
  - `menus.modifiers.Marker`
  - `menus.modifiers.AuthVisibility`
  - **menus.modifiers.Level:**

    `menus.modifiers.Level.mark_levels()`

* return the nodes to the context in the variable `children`

5.3 Publishing

Each page in the CMS exists in two versions: public and draft. The staff users generally use the draft version to edit content and change settings for the pages. None of these changes are visible on the public site until the page

is published.

When a page is published, the page must also have all parent pages published in order to become available on the web site. If a parent page is not published at the moment, the page goes into a "pending" state where it will become automatically published once the parent page is published. This enables you to edit an entire subsection of the website and publishing it once all the work is complete.

5.4 Serving content in multiple languages

5.4.1 Basic concepts

django CMS has a sophisticated multilingual capability. It is able to serve content in multiple languages, with fallbacks into other languages where translations have not been provided. It also has the facility for the user to set the preferred language and so on.

How django CMS determines the user's preferred language

django CMS determines the user's language the same way Django does it.

- the language code prefix in the URL
- the language set in the session
- the language in the *django_language* cookie
- the language that the browser says its user prefers

It uses the django built in capabilities for this.

By default no session and cookie are set. If you want to enable this use the *cms.middleware.language.LanguageCookieMiddleware* to set the cookie on every request.

How django CMS determines what language to serve

Once it has identified a user's language, it will try to accommodate it using the languages set in *CMS_LANGUAGES*.

If `fallbacks` is set, and if the user's preferred language is not available for that content, it will use the fallbacks specified for the language in *CMS_LANGUAGES*.

What django CMS shows in your menus

If `hide_untranslated` is `True` (the default) then pages that aren't translated into the desired language will not appear in the menu.

Contributing to django CMS

6.1 Contributing to django CMS

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

Attention: If you think you have discovered a security issue in our code, please report it **privately**, by emailing us at security@django-cms.org.

Please don't raise it on:

- IRC
- GitHub
- either of our email lists

or in any other public forum until we have had a chance to deal with it.

6.1.1 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.

6.1.2 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.

6.1.3 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.

And at any point in that process, you can add: *discuss discuss discuss*, because it's always useful for everyone to pass ideas around and look at thngs together.

Running and writing tests is really important.

We have an IRC channel, our django-cms-developers email list, and of course the code reviews mechanism on GitHub - do use them.

6.1.4 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 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 could discuss how it would bring more people to the project by having a Klingon or some other translation, but that's a problem we will confront once we already have 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 for section marking:

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

6.1.5 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.

6.2 Running and writing tests

Good code needs tests.

A project like django CMS simply can't afford to incorporate new code that doesn't come with its own tests.

Tests provide some necessary minimum confidence: they can show the code will behave as it expected, and help identify what's going wrong if something breaks it.

Not insisting on good tests when code is committed is like letting a gang of teenagers without a driving licence borrow your car on a Friday night, even if you think they are very nice teenagers and they really promise to be careful.

We certainly do want your contributions and fixes, but we need your tests with them too. Otherwise, we'd be compromising our codebase.

So, you are going to have to include tests if you want to contribute. However, writing tests is not particularly difficult, and there are plenty of examples to crib from in the code to help you.

6.2.1 Running tests

There's more than one way to do this, but here's one to help you get started:

```
# create a virtual environment
virtualenv test-django-cms
# activate it
cd test-django-cms/
```

```
source bin/activate
# get django CMS from GitHub
git clone git@github.com:divio/django-cms.git
# install the dependencies for testing
# note that requirements files for other Django versions are also provided
pip install -r django-cms/test_requirements/django-1.4.txt
# run the test suite
django-cms/runtests.py
```

It can take a few minutes to run.

When you run tests against your own new code, don't forget that it's useful to repeat them for different versions of Python and Django.

6.2.2 Writing tests

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 :)

What we need

We have a wide and comprehensive library of unit-tests and integration tests with good coverage.

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.
- Easy to understand. If your test code isn't obvious, please add comments on what it's doing.

6.3 Indices and tables

- genindex
- modindex
- search

c

m

A

C

F

G

I

L

M

N

P

R

S

T

V