django cms Documentation

Release 3.1.8.dev1

Patrick Lauber

July 31, 2016

Contents

Overview

django CMS is a modern web publishing platform built with Django, the web application framework "for perfectionists with deadlines".

django CMS offers out-of-the-box support for the common features you'd expect from a CMS, but can also be easily customised and extended by developers to create a site that is tailored to their precise needs.

Web content editors looking for documentation on how to use the editing interface should refer to our Using django CMS section.

Web content developers who want to learn more about django CMS, as well as how to install, configure and customize it for their own projects will can refer to Tutorials, How-to guides, Key topics and Reference sections.

Join us online

django CMS is supported by a friendly and very knowledgeable community.

Find us:

- in our IRC channel, #django-cms, on irc.freenode.net
- on our django CMS users email list for **general** django CMS questions and discussion
- on our django CMS developers email list for discussions about the **development of django CMS**

Why django CMS?

django CMS is a well-tested CMS platform that powers sites both large and small. Here are a few of the key features:

- robust internationalisation (i18n) support for creating multilingual sites
- virtually unlimited undo history, allowing editors to revert to a previous version
- front-end editing, providing rapid access to the content management interface
- support for a variety of editors with advanced text editing features.
- a flexible plugins system that lets developers put powerful tools at the fingertips of editors, without overwhelming them with a difficult interface
- ...and much more

There are other capable Django-based CMS platforms but here's why you should consider django CMS:

- thorough documentation
- easy and comprehensive integration into existing projects - django CMS isn't a monolithic application
- a healthy, active and supportive developer community
- a strong culture of good code, including an emphasis on automated testing

Release Notes

This document refers to version 3.1.8.dev1

> **Warning:** Version 3.0 introduces some significant changes that **require** action if you are upgrading from a previous version. Please refer to *Upgrading from previous versions*

Table of contents

5.1 Tutorials

The pages in this section of the documentation are aimed at the newcomer to django CMS. They're designed to help you get started quickly, and show how easy it is to work with django CMS as a developer who wants to customise it and get it working according to their own requirements.

These tutorials take you step-by-step through some key aspects of this work. They're not intended to explain the topics in depth, or provide reference material, but they will leave you with a good idea of what it's possible to achive in just a few steps, and how to go about it.

Once you're familiar with the basics presented in these tutorials, you'll find the more in-depth coverage of the same topics in the How-to section.

The tutorials follow a logical progession, starting from installation of django CMS and the creation of a brand new project, and build on each other, so it's recommended to work through them in the order presented here.

5.1.1 Installing django CMS

We'll get started by setting up our environment.

Your working environment

We're going to assume that you have a reasonably recent version of virtualenv installed and that you have some basic familiarity with it.

Create and activate a virtual env

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

Note that if you're using Windows, to activate the virtualenv you'll need:

```
env\Scripts\activate
```

Use the django CMS installer

The django CMS installer is a helpful script that takes care of setting up a new project.

Install it:

```
pip install djangocms-installer
```

This provides you with a new command, `djangocms`.

Create a new directory to work in, and cd into it:

```
mkdir tutorial-project
cd tutorial-project
```

Run it to create a new Django project called `mysite`:

```
djangocms -p . mysite
```

> **Warning:** djangocms-installer expects directory `.` to be empty at this stage, and will check for this, and will warn if it's not. You can get it to skip the check and go ahead anyway using the `-s` flag; **note that this may overwrite existing files**.

Windows users may need to do a little extra to make sure Python files are associated correctly if that doesn't work right away:

```
assoc .py=Python.file
ftype Python.File="C:\Users\Username\workspace\demo\env\Scripts\python.exe" "%1" %*
```

For the purposes of this tutorial, it's recommended that you answer the installer's questions as follows - where our suggestions differ from the default, they're highlighted below:

- Database configuration (in URL format): sqlite://localhost/project.db
- django CMS version: stable
- Django version: stable
- Activate Django I18N / L10N setting: yes
- Install and configure reversion support: yes
- Languages to enable. Option can be provided multiple times, or as a comma separated list: **en, de**
- Optional default time zone: America/Chicago:
- Activate Django timezone support: yes
- Activate CMS permission management: yes
- Use Twitter Bootstrap Theme: **yes**
- Use custom template set: no
- Load a starting page with examples after installation: **yes**

Create a Django admin user when invited.

Start up the runserver

```
python manage.py runserver
```

Open http://localhost:8000/ in your browser, where you should be presented with your brand new django CMS homepage.

Congratulations, you now have installed a fully functional CMS!

To log in, append `?edit` to the URL and hit enter. This will enable the toolbar, from where you can log in and manage your website. Switch to `Draft` mode to add and edit content.

Try to switch between `Live` and `Draft` view, between `Structure` and `Content` mode, add plugins, move them around and delete them again.

To add a *Text* or or other plugin elements to a placeholder:

1. switch to `Structure` mode

2. select the menu icon on the placeholder's title bar

3. select a plugin type to add

5.1.2 Templates & Placeholders

In this tutorial we'll introduce Placeholders, and we're also going to show how you can make your own HTML templates CMS-ready.

Templates

You can use HTML templates to customise the look of your website, define Placeholders to mark sections for managed content and use special tags to generate menus and more.

You can define multiple templates, with different layouts or built-in components, and choose them for each page as required. A page's template can be switched for another at any time.

You'll find the site's templates in `mysite/templates`. If you didn't change the automatically-created home page's template, it's `feature.html`.

Placeholders

Placeholders are an easy way to define sections in an HTML template that will be filled with content from the database when the page is rendered. This content is edited using django CMS's frontend editing mechanism, using Django templatetags.

You can see them in `feature.html`: `{% placeholder "feature" %}` and `{% placeholder "content" %}`.

You'll also see `{% load cms_tags %}` in that file - `cms_tags` is the required templatetag library.

If you're not already familiar with Django templatetags, you can find out more in the Django documentation.

Try removing a placeholder from the template, or adding a new one in the template's HTML structure.

Static Placeholders

The content of the placeholders we've encountered so far is different for every page. Sometimes though you'll want to have a section on your website which should be the same on every single page, such as a footer block.

You *could* hardcode your footer into the template, but it would be nicer to be able to manage it through the CMS. This is what **static placeholders** are for.

Static placeholders are an easy way to display the same content on multiple locations on your website. Static placeholders act almost like normal placeholders, except for the fact that once a static placeholder is created and you added content to it, it will be saved globally. Even when you remove the static placeholders from a template, you can reuse them later.

So let's add a footer to all our pages. Since we want our footer on every single page, we should add it to our base template (`mysite/templates/base.html`). Place it at the bottom of the HTML body:

```
<footer>
  {% static_placeholder 'footer' %}
</footer>
```

Save the template and return to your browser. Change to `Draft` and then `Structure` mode and add some content to it.

After you've saved it, you'll see that it appears on your site's other pages too.

Rendering Menus In order to render the CMS's menu in your template you can use the show_menu tag.

The example we use in `mysite/templates/base.html` is:

```html
<ul class="nav navbar-nav">
    {% show_menu 0 1 100 100 "menu.html" %}
</ul>
```

Any template that uses `show_menu` must load the CMS's `menu_tags` library first:

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

If you chose "bootstrap" while setting up with djangocms-installer, the menu will already be there and `templates/menu.html` will already contain a version that uses bootstrap compatible markup.

Next we'll look at django CMS plugins.

5.1.3 Plugins

In this tutorial we're going to take a Django poll app and integrate it into the CMS.

Install the polls app

Install the application from its GitHub repository using `pip -e` - this also places it in your virtualenv's `src` directory as a cloned Git repository:

```
pip install -e git+http://git@github.com/divio/django-polls.git#egg=django-polls
```

You should end up with a folder structure similar to this:

```
env/
    src/
        django-polls/
            polls/
                __init__.py
                admin.py
                models.py
                templates/
                tests.py
                urls.py
                views.py
```

Let's add it this application to our project. Add `'polls'` to the end of `INSTALLED_APPS` in your project's *settings.py* (see the note on *The INSTALLED_APPS setting* about ordering).

Add the following line to `urlpatterns` in the project's `urls.py`:

```
url(r'^polls/', include('polls.urls', namespace='polls')),
```

Make sure this line is included **before** the line for the django-cms urls:

```
url(r'^', include('cms.urls')),
```

django CMS's URL pattern needs to be last, because it "swallows up" anything that hasn't already been matched by a previous pattern.

Now run the application's migrations:

```
python manage.py migrate polls
```

At this point you should be able to create polls and choices in the Django admin - localhost:8000/admin/ - and fill them in at `/polls/`.

However, in pages of the polls application we only have minimal templates, and no navigation or styling. Let's improve this by overriding the polls application's base template.

add `my_site/templates/polls/base.html`:

```
{% extends 'base.html' %}

{% block content %}
    {% block polls_content %}
    {% endblock %}
{% endblock %}
```

Open the `/polls/` again. The navigation should be visible now.

So now we have integrated the standard polls app in our project. But we've not done anything django CMS specific yet.

Creating a plugin

If you've played around with the CMS for a little, you've probably already encountered CMS Plugins. They are the objects you can place into placeholders on your pages through the frontend: "Text", "Image" and so forth.

We're now going to extend the django poll app so we can embed a poll easily into any CMS page. We'll put this integration code in a separate package in our project.

This allows integrating 3rd party apps without having to fork them. It would also be possible to add this code directly into the django-polls app to make it integrate out of the box.

Create a new package at the project root called `polls_plugin`:

> python manage.py startapp polls_plugin

So our workspace looks like this:

```
env/
    src/  # the django polls application is in here
polls_plugin/  # the newly-created application
    __init__.py
    admin.py
    models.py
    tests.py
    views.py
my_site/
static/
project.db
requirements.txt
```

The Plugin Model

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

```python
from django.db import models
from cms.models import CMSPlugin
from polls.models import Poll


class PollPlugin(CMSPlugin):
    poll = models.ForeignKey(Poll)

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

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

The Plugin Class

Now create a file `cms_plugins.py` in the same folder your models.py is in. The plugin class is responsible for providing django CMS with the necessary information to render your plugin.

For our poll plugin, we're going to write the following plugin class:

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


class CMSPollPlugin(CMSPluginBase):
    model = PollPlugin  # model where plugin data are saved
    module = _("Polls")
    name = _("Poll Plugin")  # name of the plugin in the interface
    render_template = "djangocms_polls/poll_plugin.html"

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

plugin_pool.register_plugin(CMSPollPlugin)  # 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 convention for plugin naming is as follows:

- SomePlugin: the *model* class
- CMSSomePlugin: the *plugin* class

You don't need to follow this, but it's a sensible thing to do.

The template

The `render_template` attribute in the plugin class is required, and tells the plugin which *render_template* to use when rendering.

In this case the template needs to be at `polls_plugin/templates/djangocms_polls/poll_plugin.html` and should look something like this:

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

<form action="{% url 'polls:vote' instance.poll.id %}" method="post">
    {% csrf_token %}
    <div class="form-group">
        {% for choice in instance.poll.choice_set.all %}
            <div class="radio">
                <label>
                    <input type="radio" name="choice" value="{{ choice.id }}">
                    {{ choice.choice_text }}
                </label>
            </div>
        {% endfor %}
    </div>
    <input type="submit" value="Vote" />
</form>
```

Now add `polls_plugin` to `INSTALLED_APPS` and create a database migration to add the plugin table:

```
python manage.py makemigrations polls_plugin
python manage.py migrate polls_plugin
```

Finally, start the runserver and visit http://localhost:8000/.

You can now drop the `Poll Plugin` into any placeholder on any page, just as you would any other plugin.

Next we'll integrate the Polls application more fully into our django CMS project.

5.1.4 Apphooks

Right now, our django Polls app is statically hooked into the project's `urls.py`. This is all right, but we can do more, by attaching applications to django CMS pages.

We do this with an **Apphook**, created using a `CMSApp` subclass, which tells the CMS how to include that app.

Apphooks live in a file called `cms_app.py`, so create one in your Poll application.

This is the most basic example for a django CMS app:

```python
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)
    app_name = "polls"

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

You'll need to restart the runserver to allow the new apphook to become available.

In the admin, create a new child page of the Home page. In its *Advanced settings*, choose "Polls App" from the *Application* menu, and Save.

Refresh the page, and you'll find that the Polls application is now available directly from the new django CMS page. (Apphooks won't take effect until the server has restarted, though this is not generally an issue on the runserver, which can handle this automatically.)

You can now remove the inclusion of the polls urls in your project's `urls.py` - it's no longer required there.

Next, we're going to install a django-CMS-compatible third-party application.

5.1.5 Integrating a third-party application

We've already written our own django CMS plugins and apps, but now we want to extend our CMS with a third party app, Aldryn News & Blog.

First, we need to install the app into our virtual environment from PyPI:

```
pip install aldryn-newsblog
```

Add the app and any of its requirements that are not there already to INSTALLED_APPS in settings.py. Some *will* be already present; it's up to you to check them:

```
'aldryn_apphooks_config',
'aldryn_boilerplates',
'aldryn_categories',
'aldryn_newsblog',
'aldryn_people',
'aldryn_reversion',
'djangocms_text_ckeditor',
'easy_thumbnails',
'filer',
'parler',
'reversion',
'sortedm2m',
'taggit',
```

One of the dependencies is django-filer. It provides a special feature that allows nicer image cropping. For this to work it needs it's own thumbnail processor (easy-thumbnails) to be inserted in settings.py:

```
THUMBNAIL_PROCESSORS = (
    'easy_thumbnails.processors.colorspace',
    'easy_thumbnails.processors.autocrop',
    # 'easy_thumbnails.processors.scale_and_crop',
    'filer.thumbnail_processors.scale_and_crop_with_subject_location',
    'easy_thumbnails.processors.filters',
)
```

aldryn-newsblog uses aldryn-boilerplates to provide multiple sets of templates and staticfiles for different css frameworks. We're using bootstrap3 in this tutorial, so lets choose *bootstrap3*.:

```
ALDRYN_BOILERPLATE_NAME='bootstrap3'
```

Add boilerplates finder to STATICFILES_FINDERS:

```
STATICFILES_FINDERS = [
    'django.contrib.staticfiles.finders.FileSystemFinder',
    # important! place right before django.contrib.staticfiles.finders.AppDirectoriesFinder
    'aldryn_boilerplates.staticfile_finders.AppDirectoriesFinder',
    'django.contrib.staticfiles.finders.AppDirectoriesFinder',
]
```

If STATICFILES_FINDERS is not defined in your settings.py just copy and paste the above code.

Add the boilerplate template loader to TEMPLATE_LOADERS:

```
TEMPLATE_LOADERS = (
    'django.template.loaders.filesystem.Loader',
    'aldryn_boilerplates.template_loaders.AppDirectoriesLoader',
    'django.template.loaders.app_directories.Loader',
    'django.template.loaders.eggs.Loader'
)
```

Add the boilerplates context processor to TEMPLATE_CONTEXT_PROCESSORS:

```
TEMPLATE_CONTEXT_PROCESSORS = [
    # ...
    'aldryn_boilerplates.context_processors.boilerplate',
]
```

Since we added a new app, we need to update our database:

```
python manage.py migrate
```

Start the server again.

The newsblog application comes with a django CMS apphook, so add a new django CMS page (let's call it 'Blog'), and add the blog application to it as you did for Polls in the previous tutorial step. In this case we also have to add an "Application configuration" (see the field right under the apphook field). You can configure some settings here, like the url format. It's also possible to add multiple instances of the application, if you like. The *Instance namespace* should be `blog` (this is used for reversing urls). Choose `Blog` as the *Application title* and choose whatever *Permalink type* you prefer.

Publish the new page, and you should find the blog application at work there.

You may need to restart your server at this point.

You can add new blog posts using the admin, but also have a look at the toolbar. When you're within the urls of the blog, you should see an extra menu item called "Blog". You can now select "Blog" > "Add new article..." from it and add a new blog post directly from there.

Try also inserting a "Latest articles" plugin into another page - as a good django CMS application, *Aldryn News & Blog* comes with plugins.

In the next tutorial, we're going to integrate our Polls app into the toolbar in, just like the blog application has been.

5.1.6 Extending the Toolbar

django CMS allows you to control what appears in the toolbar. This allows you to integrate your application in the frontend editing mode of django CMS and provide your users with a streamlined editing experience.

Registering Toolbar items

There are two ways to control what gets shown in the toolbar.

One is the `CMS_TOOLBARS` setting. This gives you full control over which classes are loaded, but requires that you specify them all manually.

The other is to provide `cms_toolbar.py` files in your apps, which will be automatically loaded as long `CMS_TOOLBARS` is not set (or set to *None*). We'll work with this second method.

Create a new `cms_toolbar.py` file in your Polls application (NOTE: *not* in the Polls Plugin application we were working with in the previous tutorial):

```python
from django.utils.translation import ugettext_lazy as _
from cms.toolbar_pool import toolbar_pool
from cms.toolbar_base import CMSToolbar
from cms.utils.urlutils import admin_reverse
from polls.models import Poll


@toolbar_pool.register
class PollToolbar(CMSToolbar):
    supported_apps = (
        'polls',
        'polls_plugin',
    )
    watch_models = [Poll,]

    def populate(self):
        if not self.is_current_app:
            return

        menu = self.toolbar.get_or_create_menu('poll-app', _('Polls'))
```

```
        menu.add_sideframe_item(
            name=_('Poll list'),
            url=admin_reverse('polls_poll_changelist'),
        )

        menu.add_modal_item(
            name=_('Add new poll'),
            url=admin_reverse('polls_poll_add'),
        )
```

What we're doing above is this:

- defining a `CMSToolbar` subclass

- registering the toolbar class with `@toolbar_pool.register`

- defining a `populate()` method that adds an item to the menu

The `populate()` method:

- checks whether we're in a page belonging to this application

- if so, it creates a menu if one's not already there

- adds a menu item to list all polls as a sideframe

- adds a menu item to add a now poll as a modal window

Your `cms_toolbar.py` file should contain classes that extend `cms.toolbar_base.CMSToolbar` and are registered using `cms.toolbar_pool.toolbar_pool.register()`. The register function can be used as a decorator.

`CMSToolbar` instances will have these attributes:

- `toolbar`: the toolbar object

- `request` the current request

- `is_current_app` a flag indicating whether the current request is handled by the same app as the function is in (use `supported_apps` to add more apps that should be considered the "active app")

- `app_path` the name of the app used for the current request

- `watch_models` which allow the frontend editor to redirect the user to the model instance `get_absolute_url` whenever an instance is created or saved through the frontend editor (see *Detecting url changes* for details)

`CMSToolbar` subclasses must implement a `populate` method. The `populate` method will only be called if the current user is a staff user. `supported_apps` is a list of app names that should be considered as `is_current_app`. Usually you don't need to set `supported_apps`, but in our case we need it so `is_current_app` can be detected properly (because the views for the poll app are in `polls` and our `cms_toolbar.py` is in the `polls_plugin` app).

There's a lot more to django CMS toolbar classes than this - see Extending the Toolbar for more.

5.1.7 Extending the navigation menu

You may have noticed that while our Polls application has been integrated into the CMS, with plugins, toolbar menu items and so on, the site's navigation menu is still only determined by django CMS Pages.

We can hook into the django CMS menu system to add our own nodes to that navigation menu.

For this we need a file called `menu.py` in our application. Add `polls_plugin/menu.py`:

```python
from django.core.urlresolvers import reverse
from django.utils.translation import ugettext_lazy as _
```

```python
from cms.menu_bases import CMSAttachMenu
from menus.base import NavigationNode
from menus.menu_pool import menu_pool

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():
            node = NavigationNode(
                title=poll.question,
                url=reverse('polls:detail', args=(poll.pk,)),
                id=poll.pk,  # unique id for this node within the menu
            )
            nodes.append(node)
        return nodes

menu_pool.register_menu(PollsMenu)
```

What's happening here:

- we define a `PollsMenu` class, and register it
- we give the class a `name` attribute (will be displayed in admin)
- in its `get_nodes()` method, we build and return a list of nodes, where:
- first we get all the `Poll` objects
- ... and then create a `NavigationNode` object from each one
- ... and return a list of these `NavigationNodes`

This menu class is not active until attached to the apphook we created earlier. So open your `cms_app.py` and add:

```python
from polls_plugin.menu import PollsMenu
```

for importing `PollsMenu` and

> menus = [PollsMenu]

to the `PollsApp` class.

Any page that is attached to the `Polls` application will now have submenu items for each of the Polls in the database.

5.1.8 Namespaced Apphooks

Namespaced configuration for Apphooks allows to have multiple instances of the same app be used in different locations in the page tree. This also provides the building blocks needed to have some extra configuration in the database to control some aspects of each instance of the app.

Basic concepts

The concept of apphooks-config is to store all the configuration in an applications-specific model, and let the developer specify the desired options in a form. In the views the configuration model instance specific for the

current application namespace is loaded (through a mixin) and thus it is available in the view to provide the configuration for the current namespace.

Namespaces can be created on the fly in the Page admin Advanced settings.

When creating an application configuration, you are in fact defining a namespace, which is saved in the same field in the Page model as the plain namespaces.

step-by-step implementation

We're going to create a new application called FAQ. It is a simple list of Frequently asked questions. And we'll make it possible to setup multiple sets of FAQ Entries at different locations of the pagetree, each with its individual set of entries.

Lets create our new FAQ app:

```
python manage.py startapp faq
```

`models.py`:

```python
from aldryn_apphooks_config.fields import AppHookConfigField
from aldryn_apphooks_config.managers import AppHookConfigManager
from django.db import models
from faq.cms_appconfig import FaqConfig


class Entry(models.Model):
    app_config = AppHookConfigField(FaqConfig)
    question = models.TextField(blank=True, default='')
    answer = models.TextField()

    objects = AppHookConfigManager()

    def __unicode__(self):
        return self.question

    class Meta:
        verbose_name_plural = 'entries'
```

The `app_config` field is essentially a `ForeignKey` to a model we'll define in the next step. That model will hold the specific namespace configuration and allows to assign an Entry to a namespace.

The custom `AppHookConfigManager` simply makes the default queryset easily filterable by the namespace like this: `Entry.objects.namespace('foobar')`.

Next lets define the AppHookConfig model (in `cms_appconfig.py`):

```python
from aldryn_apphooks_config.models import AppHookConfig
from aldryn_apphooks_config.utils import setup_config
from app_data import AppDataForm
from django.db import models
from django import forms
from django.utils.translation import ugettext_lazy as _


class FaqConfig(AppHookConfig):
    paginate_by = models.PositiveIntegerField(
        _('Paginate size'),
        blank=False,
        default=5,
    )


class FaqConfigForm(AppDataForm):
```

```
    title = forms.CharField()
setup_config(FaqConfigForm, FaqConfig)
```

The implementation can be completely empty as the minimal schema is defined in the parent (abstract) model.

In this case we're defining a few extra fields though. We're defining `paginate_by` as a normal model field. We'll use it later to control how maye entries should be displayed per page. For the title, we're using a `AppDataForm` (see django-appdata). These forms can also be extended from other applications by just registering them. So other apps can add fields without altering the model (it's saved in a json field). The title field could also just be a model field, like `paginate_by`. But we're using the AppDataForm to demonstrate this capability.

In `admin.py` we need to define all fields we'd like to display:

```python
from django.contrib import admin
from .cms_appconfig import FaqConfig
from .models import Entry
from aldryn_apphooks_config.admin import ModelAppHookConfig, BaseAppHookConfig


class EntryAdmin(ModelAppHookConfig, admin.ModelAdmin):
    list_display = (
        'question',
        'answer',
        'app_config',
    )
    list_filter = (
        'app_config',
    )
admin.site.register(Entry, EntryAdmin)


class FaqConfigAdmin(BaseAppHookConfig, admin.ModelAdmin):
    def get_config_fields(self):
        return (
            'paginate_by',
            'config.title',
        )
admin.site.register(FaqConfig, FaqConfigAdmin)
```

`get_config_fields` defines the fields that should be displayed. Any fields using the AppData forms need to be prefixed by `config.`.

Now lets create the apphook with appconfig support (`cms_app.py`):

```python
from aldryn_apphooks_config.app_base import CMSConfigApp
from cms.apphook_pool import apphook_pool
from django.utils.translation import ugettext_lazy as _
from .cms_appconfig import FaqConfig


class FaqApp(CMSConfigApp):
    name = _("Faq App")
    urls = ["faq.urls"]
    app_name = "faq"
    app_config = FaqConfig

apphook_pool.register(FaqApp)
```

We have all the basics in place. Now we'll add a list view for the FAQ entries that only displays entries for the currently used namespace (`views.py`):

```python
from aldryn_apphooks_config.mixins import AppConfigMixin
from django.views import generic
from .models import Entry
```

```python
class IndexView(AppConfigMixin, generic.ListView):
    model = Entry
    template_name = 'faq/index.html'

    def get_queryset(self):
        qs = super(IndexView, self).get_queryset()
        return qs.namespace(self.namespace)

    def get_paginate_by(self, queryset):
        try:
            return self.config.paginate_by
        except AttributeError:
            return 10
```

AppConfigMixin provides a complete support to namespaces, so the view is not required to set anything specific to support them; the following attributes are set for the view class instance:

- current namespace in `self.namespace`

- namespace configuration (the instance of NewsBlogConfig) in `self.config`

- current application in the `current_app parameter` passed to the `Response` class

In this case we're filtering to only show entries assigned to the current namespace in `get_queryset`. There is no magic behind `qs.namespace`, it could have also been written as `qs.filter(app_config__namespace=self.namespace)`.

In `get_paginate_by` we use the value from our appconfig model.

And now for the rest of the missing files of the FAQ app.

And the template (`faq/templates/faq/index.html`):

```html
{% extends 'base.html' %}

{% block content %}
    <h1>{{ view.config.title }}</h1>
    <p>Namespace: {{ view.namespace }}</p>
    <dl>
        {% for entry in object_list %}
            <dt>{{ entry.question }}</dt>
            <dd>{{ entry.answer }}</dd>
        {% endfor %}
    </dl>

    {% if is_paginated %}
        <div class="pagination">
            <span class="step-links">
                {% if page_obj.has_previous %}
                    <a href="?page={{ page_obj.previous_page_number }}">previous</a>
                {% else %}
                    previous
                {% endif %}

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

                {% if page_obj.has_next %}
                    <a href="?page={{ page_obj.next_page_number }}">next</a>
                {% else %}
                    next
                {% endif %}
            </span>
```

```html
        </div>
    {% endif %}
{% endblock %}
```

`urls.py`:

```python
from django.conf.urls import patterns, url
from . import views


urlpatterns = patterns('',
    url(r'^$', views.IndexView.as_view(), name='index'),
)
```

Finally, lets add `faq` to `INSTALLED_APPS` and create a migrations:

```
python manage.py makemigrations faq
python manage.py migrate faq
```

Now we should be all set. Create two pages with the faq apphook with different namespaces and different configurations. Also create some entries assigned to the two namespaces. Don't forget to publish the pages with the apphook and restart the server.

5.1.9 Contributing

Note: The topics of this chapter is covered in more details into the *Contributing to django CMS* and *Running and writing tests* sections of the documentation

django CMS is an open source and open project which welcomes anyone who want to provide its contribution at any level of knowledge.

You can contribute code, *documentation* and *translations*.

Starting contributing code

Before actually coding anything, please read the *guidelines* and the *policies* regulating the django CMS project development. Adhering to them will make much easier for the core developers to validate and accept your contribution.

The basic step to contribute are:

1. fork django CMS project repostory on github

2. clone your fork on your local computer:

```
git clone git@github.com:YOUR_USERNAME/django-cms.git
```

3. create a virtualenv:

```
virtualenv cms-develop
source cms-develop/bin/activate
```

4. install the dependencies:

```
cd django-cms
pip install -r test_requirements/django-1.7.txt
```

or whichever Django version you target

5. code you contribution

6. run the tests:

```
python develop.py test
```

7. commit and push your code into a feature branch:

```
git checkout -b my_fix
git commit
git push origin my_fix
```

8. open a pull request on github

Target branches

At one point in time django CMS project will have at least two active branches:

- latest `support/version.x` which you sholud target if you submit bugfixes for `version.x`

- `develop` for new features and bugfixes for latest version if a corresponding `support/version.y` does not exists (yet)

How to write a test

django CMS test suite contains a mix of unit tests, functional tests, regression tests and integration tests.

Depending on your contribution, you will write a mix of them.

Let's start with something simple.

Let's say you want to test the behavior of `CMSPluginBase.render` method:

```python
class CMSPluginBase(six.with_metaclass(CMSPluginBaseMetaclass, admin.ModelAdmin)):

    ...

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

Writing a unit test for it will require us to test whether the return context object contains the declared attributes with the correct values.

We will start with a new class in an existing django CMS test module (`cms.tests.plugins` in this case):

```python
class SimplePluginTestCase(CMSTestCase):
    pass
```

Let's try to run it (given you've setup correctly your environment as in *start-contributing*:

```
python develop.py test cms.SimplePluginTestCase
```

This will call the new test case class only and it's hany when creating new tests and iterating quickly throught the steps. A full test run (`python develop.py test`) is required before opening a pull request.

This is the output you'll get:

```
Creating test database for alias 'default'...

----------------------------------------------------------------
Ran 0 tests in 0.000s

OK
```

Which is correct as we have no test in our test case. Let's add and (empty) one:

```python
class SimplePluginTestCase(CMSTestCase):

    def test_render_method(self):
        pass
```

Running the test command again will return a sighltly different output:

```
Creating test database for alias 'default'...
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

This looks better, but it's not that meaningful as we're not testing anything.

Write a real test:

```python
class SimplePluginTestCase(CMSTestCase):

    def test_render_method(self):
        """
        Tests the CMSPluginBase.render method by checking that the appropriate variables
        are set in the returned context
        """
        from cms.api import create_page
        my_page = create_page('home', language='en', template='col_two.html')
        placeholder = my_page.placeholders.get(slot='col_left')
        context = self.get_context('/', page=my_page)
        plugin = CMSPluginBase()

        new_context = plugin.render(context, None, placeholder)
        self.assertTrue('placeholder' in new_context)
        self.assertEqual(placeholder, context['placeholder'])
        self.assertTrue('instance' in new_context)
        self.assertIsNone(new_context['instance'])
```

and run it:

```
Creating test database for alias 'default'...
.
----------------------------------------------------------------------
Ran 1 test in 0.044s

OK
```

Output is quite similar than the previous run, only the longer execution time gives us a hint that this test is actually doing something.

Let's quickly check the test code.

To test `CMSPluginBase.render` method we need a RequestContext instance and a placeholder. As `CMSPluginBase` does not have any *configuration model*, the instance argument can be `None`.

1. Create a page instance to get the placeholder

2. Get the placeholder by filtering the placeholders of the page instance on the expected placeholder name

3. Create a context instance by using the provided super class method

4. Call the render method on a CMSPluginBase instance; being stateless, it's easy to call `render` of a bare instance of the `CMSPluginBase` class, which helps in tests

5. Assert a few things the method must provide on the returned context instance

As you see, even a simple test like this assumes and uses many feature of the test utils provided by django CMS. Before attempting to write a test, take your time to explore the content of `cms.test_utils` pack-

---

age and check the shipped templates, example applications and, most of all, the base testcases defined in `cms.test_utils.testscases` which provide *a lot* of useful methods to prepare the environment for our tests or to create useful test data.

Submitting your code

After the code and the tests are ready and packed in commits, you must submit it for review and merge in the django CMS github project.

As stated above, always create a feature branch for your code, being it a fix or a new feature; then you can create a Pull Request from your branch to the *target branch* on django CMS.

Acceptance criteria

Matching this criteria from the very beginning will help the core developers to be able to review your submission more quickly and efficiently and will greatly help your code to be merged in.

Features   To be accepted, proposed features should have *at least*:

- natural language documentation in the `docs` folder describing the feature, its usage and potentially backward incompatibilities.
- inline documentation (comments and docstrings) in the critical areas of the code explaining the behavior
- appropriate test coverage
- Python 2/3 compatibility
- South and Django migrations (where applicable)

The pull request description must briefly describe the feature and the intended goal and benefits.

Bugs   To be accepted, proposed bug fixes should have *at least*:

- inline documentation (comments and docstrings) in the critical areas of the code explaining the behavior
- at least 1 regression test that demonstrates the issue and the fix
- Python 2/3 compatibility
- South and Django migrations (where applicable)

The pull request description must briefly describe the bug and the steps for its solution; in case the bug has been opened elsewhere, it must be linked in the Pull Request description, describing the fix.

If you want to install django CMS into an existing project, or prefer to configure django CMS by hand, rather than using the automated installer, see Installing django CMS by hand and then follow the rest of the tutorials.

Either way, you'll be able to find support and help from the numerous friendly members of the django CMS community, either on our mailinglist or IRC channel `#django-cms` on the `irc.freenode.net` network.

5.2 How-to guides

These guides presuppose some familiarity with django CMS. They cover some of the same territory as the Tutorials, but in more detail.

5.2.1 Installing django CMS by hand

This is how to install django CMS 'the hard way' (it's not really that hard, but there is an easier way).

It's suitable if you want to dive in to integrating django CMS into an existing project, are already experienced at setting up Django projects or indeed like to do things the hard way.

If you prefer an easier way using an automated configuration tool - definitely recommended for new users - see Installing django CMS, which is part of a complete introductory tutorial.

This document assumes you are familiar with Python and Django. After you've integrated django CMS into your project, you should be able to follow the Tutorials.

Requirements

- Python 2.6, 2.7, 3.3 or 3.4.
- Django 1.6.x, 1.7.x, 1.8.x
- South 1.0.1 or higher (Only required up to Django 1.6)
- django-classy-tags 0.6.2 or higher
- django-treebeard 3.0
- django-sekizai 0.8.2 or higher
- djangocms-admin-style
- An installed and working instance of one of the databases listed in the *Databases* section.

Note: When installing the django CMS using pip, all of the dependencies will be installed automatically.

Recommended

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

Text Editors

- Django CMS CKEditor for a WYSIWYG editor 2.4.0 or higher

Other Plugins

- djangocms-link
- djangocms-snippet
- djangocms-style
- djangocms-column
- djangocms-grid
- djangocms-oembed
- djangocms-table

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 1.8.X (with Django 1.6.X and Django 1.7.X) 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 3.0.x, only the most recent 10 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.

Installing

Installing in a virtualenv using pip

Installing inside a virtualenv is the preferred way to install any Django installation.

```
sudo pip install --upgrade virtualenv
virtualenv env
```

Note: If you are *not* using a system-wide install of Python (such as with Homebrew), omit the usage of `sudo` when installing via `pip`.

Switch to the virtualenv at the command line by typing:

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

Next, install the CMS:

```
pip install django-cms
```

This will automatically install all of the *requirements* listed above.

While you could install packages one at a time using pip, 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>=3.0

# These dependencies are brought in by django CMS, but if you want to
# lock-in their version, specify them
Django>=1.7

django-treebeard==3.0
django-sekizai==0.8.2
django-classy-tags==0.6.2
djangocms-admin-style==0.2.2
six==1.3.0

# Optional, recommended packages
Pillow>=2
django-filer==0.9.9
```

```
cmsplugin-filer==0.10.1
django-reversion==1.8.5
```

Note: In the above list, packages are pinned to specific version as an example; those are not mandatory versions; refer to *requirements* for any version-specific restrictions.

If you are using PostgreSQL as your database, add the Python adapter to your requirements file:

```
psycopg2
```

For MySQL you would instead add:

```
mysql-python
```

Note: While the django CMS is compatible with Python 3.3+, the `mysql-python` package is not.

Before you install the Python adapters for your chosen database, you will need to first install the appropriate headers and development libraries. See the platform specific notes below.

Installing on Ubuntu

If you're using Ubuntu (tested with 14.04), the following should get you started:

```
sudo aptitude install python-pip
sudo pip install virtualenv
```

Next, install the appropriate libraries to build the Python adapters for your selected database. For PostgreSQL:

```
sudo aptitude install libpq-dev postgresql-client-9.3 python-dev
```

For MySQL:

```
sudo aptitude install libmysqlclient-dev python-dev
```

Installing and configuring database servers are beyond the scope of this document. See *Databases* below for more information and related links.

Installing on Mac OSX

If you are using the system provided Python (2.6 or later), ensure you have `pip` installed.

```
sudo easy_install pip
sudo pip install virtualenv
```

If you're using Homebrew you can install pip and virtualenv with the python generic package:

```
brew install python
pip install virtualenv
```

Next, install the appropriate libraries to build the Python adapters for your selected database. For PostgreSQL:

```
brew install postgres
```

For MySQL:

```
brew install mysql
```

Note: Homebrew does not set the databases to run automatically. The software necessary for the Python adapters will be installed but if you wish to run the database server locally, follow the Homebrew instructions shown in the terminal output after installing.

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.

Configuration and setup

Preparing the environment

The following steps assume your Django project will be - or already is - in `~/workspace/myproject`, and that you'll be using a virtualenv.

If you already have a virtualenv with a project in it, activate it and move on to *Configuring your project for django CMS*.

Otherwise:

```
cd ~/workspace/myproject/
virtualenv env
source env/bin/activate
pip install -r requirements.txt
```

Create a new Django project

```
django-admin.py startproject myproject
```

If this is new to you, you ought to read the [official Django tutorial](#), which covers starting a new project.

Configuring your project for django CMS

Open the `settings.py` file in your project.

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

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

Add the following apps to your `INSTALLED_APPS`. This includes django CMS itself as well as its dependencies and other highly recommended applications/libraries:

```python
'cms',  # django CMS itself
'treebeard',  # utilities for implementing a tree
'menus',  # helper for model independent hierarchical website navigation
'south',  # Only needed for Django < 1.7
'sekizai',  # for javascript and css management
```

```
'djangocms_admin_style',   # for the admin skin. You **must** add 'djangocms_admin_style' in the l
'django.contrib.messages',   # to enable messages framework (see :ref:`Enable messages <enable-mes
```

Also add any (or all) of the following plugins, depending on your needs (see the note in *The INSTALLED_APPS setting* about ordering):

```
'djangocms_file',
'djangocms_flash',
'djangocms_googlemap',
'djangocms_inherit',
'djangocms_picture',
'djangocms_teaser',
'djangocms_video',
'djangocms_link',
'djangocms_snippet',
```

Note: Most of the above plugins were previously distributed with django CMS, however, most of them are now located in their own repositories and renamed. Furthermore plugins: 'cms.plugins.text' and 'cms.plugins.twitter' have been removed from the django CMS bundle. Read *3.0 release notes* for detailed information.

Warning: Adding the 'djangocms_snippet' plugin is a potential security hazard. For more information, refer to snippet_plugin.

Some commonly-used plugins are described in more detail in Some commonly-used plugins. 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 djangocms_file, djangocms_picture, djangocms_teaser and djangocms_video core plugins. In this case you should check the django-filer documentation and django CMS plugin documentation for detailed installation information, and then return to this tutorial.

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.common.CommonMiddleware',
    'cms.middleware.user.CurrentUserMiddleware',
    'cms.middleware.page.CurrentPageMiddleware',
    '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.contrib.messages.context_processors.messages',
```

```
    'django.core.context_processors.i18n',
    'django.core.context_processors.request',
    'django.core.context_processors.media',
    'django.core.context_processors.static',
    'sekizai.context_processors.sekizai',
    'cms.context_processors.cms_settings',
)
```

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

Warning: Be sure to have 'django.contrib.sites' in INSTALLED_APPS and set SITE_ID parameter in your settings: they may be missing from the settings file generated by django-admin depending on your Django version and project template.

Changed in version 3.0.0.

Warning: Django messages framework is now **required** for the toolbar to work properly.
To enable it you must be check the following settings:

- INSTALLED_APPS: must contain 'django.contrib.messages'
- MIDDLEWARE_CLASSES: must contain 'django.contrib.messages.middleware.MessageMiddleware'
- TEMPLATE_CONTEXT_PROCESSORS: must contain 'django.contrib.messages.context_processors.me

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(BASE_DIR, "static")
STATIC_URL = "/static/"
```

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

```
MEDIA_ROOT = os.path.join(BASE_DIR, "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: BASE_DIR is precisely one.
    # Life is wonderful!
    os.path.join(BASE_DIR, "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.

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(BASE_DIR, 'database.sqlite'),
    }
}
```

django CMS, as well as its plugins, supports both Django 1.7 and Django 1.6 migrations.

Since version 3.1, migrations are stored in modules compatible with Django 1.7 **and** South 1.0.2 without further configuration.

django CMS plugins are being ported to the same structure; in the meantime, on Django 1.7, you may need to specify where the migrations are situated using the `MIGRATION_MODULES` setting:

```
MIGRATION_MODULES = {
    # Add also the following modules if you're using these plugins:
    'djangocms_file': 'djangocms_file.migrations_django',
    'djangocms_flash': 'djangocms_flash.migrations_django',
    'djangocms_googlemap': 'djangocms_googlemap.migrations_django',
    'djangocms_inherit': 'djangocms_inherit.migrations_django',
    'djangocms_link': 'djangocms_link.migrations_django',
    'djangocms_picture': 'djangocms_picture.migrations_django',
    'djangocms_snippet': 'djangocms_snippet.migrations_django',
    'djangocms_teaser': 'djangocms_teaser.migrations_django',
    'djangocms_video': 'djangocms_video.migrations_django',
    'djangocms_text_ckeditor': 'djangocms_text_ckeditor.migrations_django',
}
```

Please check each plugin configuration option to see how to configure Django 1.7 support.

URL configuration

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

```python
from django.conf import settings
from django.conf.urls import include, url
from django.conf.urls.i18n import i18n_patterns
from django.conf.urls.static import static
from django.contrib import admin

admin.autodiscover()  # Not required for Django 1.7.x+

urlpatterns = i18n_patterns('',
    url(r'^admin/', include(admin.site.urls)),
    url(r'^', include('cms.urls')),
) + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

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>
      <title>{% page_attribute "page_title" %}</title>
      {% 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 library. 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

django CMS uses Django 1.7's built-in support for database migrations to manage creating and altering database tables. django CMS still offers South-style migrations for users of Django up to 1.6 but as noted above, strictly

requires South>=1.0.1 in this case.

Fresh install   If you are using Django 1.7 or later run:

```
python manage.py migrate
python manage.py createsuperuser
```

If you are using Django 1.6.x run:

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

The call to `syncdb` will prompt you to create a super user. Choose 'yes' and enter appropriate values.

Upgrade   If you are upgrading your installation of django CMS from a previous version run:

```
python manage.py syncdb # Django 1.6.x only
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 "Installation Successful" screen.
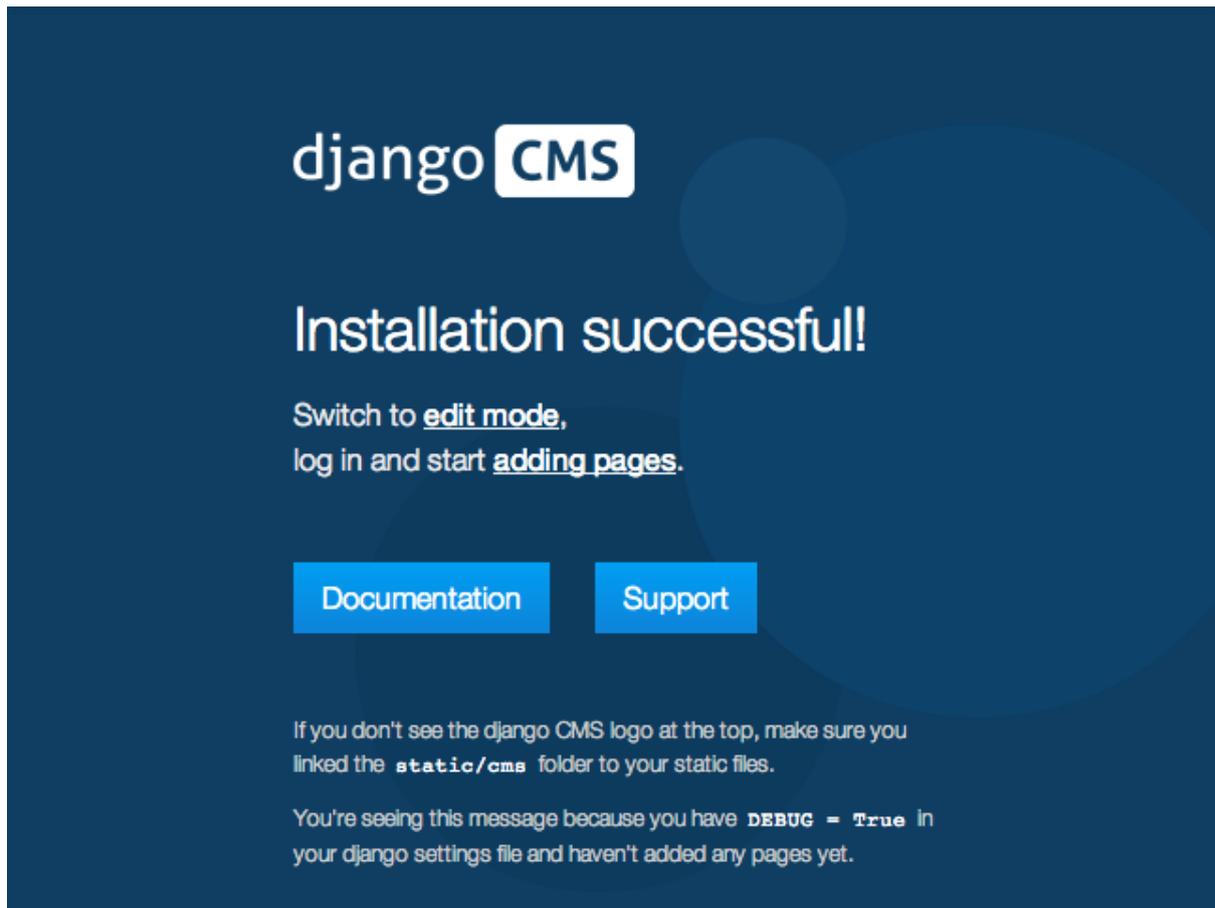
Use the new side-frame-based administration by appending '?edit' to your URL as follows: *http://127.0.0.1:8000/?edit*. This will reveal a login form.

Log in with the user you created during the database setup.

If this is your first django CMS project, read through the tutorial for a walkthrough of the main features of django CMS.

For more information on using django CMS for managing web content, see Using django CMS.

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

5.2.2 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.

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.

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`.

Because `CMSPluginBase` subclasses `ModelAdmin` several important `ModelAdmin` options are also available to CMS plugin developers. These options are often used:

- `exclude`
- `fields`
- `fieldsets`
- `form`
- `formfield_overrides`
- `inlines`
- `radio_fields`
- `raw_id_fields`
- `readonly_fields`

Please note, however, that not all `ModelAdmin` options are effective in a CMS plugin. In particular, any options that are used exclusively by the `ModelAdmin`'s `changelist` will have no effect. These and other notable options that are ignored by the CMS are:

- `actions`
- `actions_on_top`
- `actions_on_bottom`
- `actions_selection_counter`
- `date_hierarchy`
- `list_display`
- `list_display_links`
- `list_editable`
- `list_filter`
- `list_max_show_all`
- `list_per_page`
- `ordering`
- `paginator`
- `preserve_fields`
- `save_as`
- `save_on_top`
- `search_fields`
- `show_full_result_count`
- `view_on_site`

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.

The simplest plugin

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

In `cms_plugins.py`, 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"
    cache = False

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 two 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.

- `cache`: This is a property that tells the plugin rendering system in django CMS whether to cache the plugin's output to speed-up subsequent views of the same plugin. By default, the cms caches. Since we want each visitor to see output that is specific to him or her, we need to tell the cms to not cache this plugin.

And one of the following **must** be defined if `render_plugin` attribute is `True` (the default):

- `render_template`: The template to render this plugin with.

or

- `get_render_template`: A method that returns a template path to render the plugin with.

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

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 importlib import import_module
>>> m = import_module("myapp.cms_plugins")
>>> m.some_test_function()
```

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:

```python
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:

```python
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"
    cache = False

    def render(self, context, instance, placeholder):
        context = super(HelloPlugin, self).render(context, instance, placeholder)
        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:

```html
<h1>Hello {% if request.user.is_authenticated %}
  {{ request.user.first_name }} {{ request.user.last_name}}
{% else %}
  {{ instance.guest_name }}
{% endif %}</h1>
```

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:** 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.

> **Warning:** If you are using Python 2.x and overriding the `__unicode__` method of the model file, make sure to return its results as UTF8-string. Otherwise saving an instance of your plugin might fail with the frontend editor showing an <Empty> plugin instance. To return in unicode use a return statement like `return u'{0}'.format(self.guest_name)`.

Handling Relations

Everytime the page with your custom plugin is published the plugin is copied. So 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 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.

Advanced

Inline Admin

If you want to have the foreign key relation as a inline admin, you can create a admin.StackedInline class and put it in the Plugin to "inlines". Then you can use the inline Admin form for your foreign key references. inline admin:

```python
class ItemInlineAdmin(admin.StackedInline):
    model = AssociatedItem


class ArticlePlugin(CMSPluginBase):
    model = ArticlePluginModel
    name = _("Article Plugin")
    render_template = "article/index.html"
    inlines = (ItemInlineAdmin,)

    def render(self, context, instance, placeholder):
        context = super(ArticlePlugin, self).render(context, instance, placeholder)
        items = instance.associated_item.all()
        context.update({
            'items': items,
        })
        return context
```

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.js
{% addtoblock "css" %}<link rel="stylesheet" type="text/css" href="{{ MEDIA_URL }}myplugin/css/ast
{% 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

The plugin has access to the django template context. You can override variables using the `with` tag.

Example:

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

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 3 arguments:

- `instance`: The instance of the plugin model
- `placeholder`: The instance of the placeholder this plugin appears in.
- `context`: The context that is in use, including the request.

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

Example:

```python
def add_verbose_name(instance, placeholder, context):
    '''
    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`:

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

In your `yourapp.cms_plugin_processors.py`:

```python
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" place
    '''
    # 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)
```

Nested Plugins

You can nest CMS Plugins in themselves. There's a few things required to achieve this functionality:

`models.py`:

```
class ParentPlugin(CMSPlugin):
    # add your fields here

class ChildPlugin(CMSPlugin):
    # add your fields here
```

`cms_plugins.py`:

```
from .models import ParentPlugin, ChildPlugin

class ParentCMSPlugin(CMSPluginBase):
    render_template = 'parent.html'
    name = 'Parent'
    model = ParentPlugin
    allow_children = True  # This enables the parent plugin to accept child plugins
    # You can also specify a list of plugins that are accepted as children,
    # or leave it away completely to accept all
    # child_classes = ['ChildCMSPlugin']

    def render(self, context, instance, placeholder):
        context = super(ParentCMSPlugin, self).render(context, instance, placeholder)
        return context

plugin_pool.register_plugin(ParentCMSPlugin)


class ChildCMSPlugin(CMSPluginBase):
    render_template = 'child.html'
    name = 'Child'
    model = ChildPlugin
    require_parent = True  # Is it required that this plugin is a child of another plugin?
    # You can also specify a list of plugins that are accepted as parents,
    # or leave it away completely to accept all
    # parent_classes = ['ParentCMSPlugin']

    def render(self, context, instance, placeholder):
        context = super(ChildCMSPlugin, self).render(context, instance, placeholder)
        return context

plugin_pool.register_plugin(ChildCMSPlugin)
```

parent.html:

```html
{% load cms_tags %}

<div class="plugin parent">
    {% for plugin in instance.child_plugin_instances %}
        {% render_plugin plugin %}
    {% endfor %}
</div>
```

child.html:

```html
<div class="plugin child">
    {{ instance }}
</div>
```

Extending context menus of placeholders or plugins

There are three possibilities to extend the context menus of placeholders or plugins.

- You can either extend a placeholder context menu.
- You can extend all plugin context menus.
- You can extend the current plugin context menu.

For this purpose you can overwrite 3 methods on CMSPluginBase.

- *get_extra_placeholder_menu_items*
- *get_extra_global_plugin_menu_items*
- *get_extra_local_plugin_menu_items*

Example:

```python
class AliasPlugin(CMSPluginBase):
    name = _("Alias")
    allow_children = False
    model = AliasPluginModel
    render_template = "cms/plugins/alias.html"

    def render(self, context, instance, placeholder):
        context = super(AliasPlugin, self).render(context, instance, placeholder)
        if instance.plugin_id:
            plugins = instance.plugin.get_descendants(include_self=True).order_by('placeholder',
                                                                                  'position')
            plugins = downcast_plugins(plugins)
            plugins[0].parent_id = None
            plugins = build_plugin_tree(plugins)
            context['plugins'] = plugins
        if instance.alias_placeholder_id:
            content = render_placeholder(instance.alias_placeholder, context)
            print content
            context['content'] = mark_safe(content)
        return context

    def get_extra_global_plugin_menu_items(self, request, plugin):
        return [
            PluginMenuItem(
                _("Create Alias"),
                reverse("admin:cms_create_alias"),
                data={'plugin_id': plugin.pk, 'csrfmiddlewaretoken': get_token(request)},
            )
        ]
```

```python
    def get_extra_placeholder_menu_items(self, request, placeholder):
        return [
            PluginMenuItem(
                _("Create Alias"),
                reverse("admin:cms_create_alias"),
                data={'placeholder_id': placeholder.pk, 'csrfmiddlewaretoken': get_token(request)
            )
        ]

    def get_plugin_urls(self):
        urlpatterns = [
            url(r'^create_alias/$', self.create_alias, name='cms_create_alias'),
        ]
        urlpatterns = patterns('', *urlpatterns)
        return urlpatterns

    def create_alias(self, request):
        if not request.user.is_staff:
            return HttpResponseForbidden("not enough privileges")
        if not 'plugin_id' in request.POST and not 'placeholder_id' in request.POST:
            return HttpResponseBadRequest("plugin_id or placeholder_id POST parameter missing.")
        plugin = None
        placeholder = None
        if 'plugin_id' in request.POST:
            pk = request.POST['plugin_id']
            try:
                plugin = CMSPlugin.objects.get(pk=pk)
            except CMSPlugin.DoesNotExist:
                return HttpResponseBadRequest("plugin with id %s not found." % pk)
        if 'placeholder_id' in request.POST:
            pk = request.POST['placeholder_id']
            try:
                placeholder = Placeholder.objects.get(pk=pk)
            except Placeholder.DoesNotExist:
                return HttpResponseBadRequest("placeholder with id %s not found." % pk)
            if not placeholder.has_change_permission(request):
                return HttpResponseBadRequest("You do not have enough permission to alias this pla
        clipboard = request.toolbar.clipboard
        clipboard.cmsplugin_set.all().delete()
        language = request.LANGUAGE_CODE
        if plugin:
            language = plugin.language
        alias = AliasPluginModel(language=language, placeholder=clipboard, plugin_type="AliasPlug
        if plugin:
            alias.plugin = plugin
        if placeholder:
            alias.alias_placeholder = placeholder
        alias.save()
        return HttpResponse("ok")
```

Plugin data migrations

Due to the migration from mptt to treebeard in version 3.1, the plugin model is different between the two versions. Schema migration are not affected as the migration systems (both South and Django) detects the different bases.

Data migration are a different story, though.

If your datamigration does something like:

```python
MyPlugin = apps.get_model('my_app', 'MyPlugin')
```

```
for plugin in MyPlugin.objects.all():
    ... do something ...
```

You may end up with an error like `django.db.utils.OperationalError: (1054, "Unknown column 'cms_cmsplugin.level' in 'field list'")` because depending on the order the migrations are executed, the historical models may be out of sync with the applied database schema.

To keep compatibility with 3.0 and 3.x you can force the datamigration to run before the django CMS migration that creates treebeard fields, by doing this the datamigration will always be executed on the "old" database schema and no conflict will exist.

For South migrations add this:

```
from distutils.version import LooseVersion
import cms
USES_TREEBEARD = LooseVersion(cms.__version__) >= LooseVersion('3.1')


class Migration(DataMigration):

    if USES_TREEBEARD:
        needed_by = [
            ('cms', '0070_auto__add_field_cmsplugin_path__add_field_cmsplugin_depth__add_field_c')
        ]
```

For Django migrations add this:

```
from distutils.version import LooseVersion
import cms
USES_TREEBEARD = LooseVersion(cms.__version__) >= LooseVersion('3.1')


class Migration(migrations.Migration):

    if USES_TREEBEARD:
        run_before = [
            ('cms', '0004_auto_20140924_1038')
        ]
```

5.2.3 Customising navigation menus

In this document we discuss three different way of customising the navigation menus of django CMS sites.

1. *Menus*: Statically extend the menu entries

2. *Attach Menus*: Attach your menu to a page.

3. *Navigation Modifiers*: Modify the whole menu tree

Menus

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

```python
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 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.

Customize menus at runtime

To adapt your menus according to request dependent conditions (say: anonymous / logged in user), you can use *Navigation Modifiers* or you can leverage existing ones.

For example it's possible to add {'visible_for_anonymous': False} / {'visible_for_authenticated': False} attributes recognized by the django CMS core AuthVisibility modifier.

Complete example:

```
class UserMenu(Menu):
    def get_nodes(self, request):
            return [
                NavigationNode(_("Profile"), reverse(profile), 1, attr={'visible_for_anonymous': F
                NavigationNode(_("Log in"), reverse(login), 3, attr={'visible_for_authenticated':
                NavigationNode(_("Sign up"), reverse(logout), 4, attr={'visible_for_authenticated
                NavigationNode(_("Log out"), reverse(logout), 2, attr={'visible_for_anonymous': Fa
            ]
```

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.

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:

```python
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:

```python
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)
```

5.2.4 Apphooks

An **Apphook** allows you to attach a Django application to a page. For example, you might have a news application that you'd like integrated with django CMS. In this case, you can create a normal django CMS page without any content of its own, and attach the news application to the page; the news application's content will be delivered at the page's URL.

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

```python
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:** Whenever you add or remove an apphook, change the slug of a page containing an apphook or the slug if a page which has a descendant with an apphook, you have to restart your server to re-load the URL caches.
> An apphook won't appear until it is published. Take note that this also means all parent pages must also be published.

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:

```python
from django.conf.urls 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: CMS pages **below** the page to which the apphook is attached to, **can** be visible, provided that the apphook urlconf regexps are not too greedy. From a URL resolution perspective, attaching an apphook works in same way than inserting the apphook urlconf in the root urlconf at the same path as the page is attached to.

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:

```python
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:

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

```python
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:

```python
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 apphook:

```python
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.

Attaching an application multiple times

If you want to attach an application multiple times to different pages you have 2 possibilities.

1. Give every application its own namespace in the advanced settings of a page.

2. Define an `app_name` attribute on the CMSApp class.

The problem is that if you only define a namespace you need to have multiple templates per attached app.

For example:

```
{% url 'my_view' %}
```

Will not work anymore when you namespace an app. You will need to do something like:

```
{% url 'my_namespace:my_view' %}
```

The problem is now if you attach apps to multiple pages your namespace will change. The solution for this problem are application 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)
```

Note: If you do provide an `app_label`, then you will need to also give the app a unique namespace in the advanced settings of the page. If you do not, and no other instance of the app exists using it, then the 'default instance namespace' will be automatically set for you. You can then either reverse for the namespace(to target different apps) or the app_name (to target links inside the same app).

If you use app namespace you will need to give all your view `context` a `current_app`:

```
def my_view(request):
    current_app = resolve(request.path_info).namespace
    context = RequestContext(request, current_app=current_app)
    return render_to_response("my_template.html", context_instance=context)
```

Note: You need to set the current_app explicitly in all your view contexts as django does not allow an other way of doing this.

You can reverse namespaced apps similarly and it "knows" in which app instance it is:

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

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

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

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):
    current_app = resolve(request.path_info).namespace

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

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

```python
class MyPlugin(CMSPluginBase):
    def render(self, context, instance, placeholder):
        # ...
        current_app = resolve(request.path_info).namespace
        reversed_url = reverse('myapp_namespace:app_main',
                current_app=current_app)
        # ...
```

Apphook permissions

By default all apphooks have the same permissions set as the page they are assigned to. So if you set login required on page the attached apphook and all it's urls have the same requirements.

To disable this behavior set `permissions = False` on your apphook:

```python
class SampleApp(CMSApp):
    name = _("Sample App")
    urls = ["project.sampleapp.urls"]
    permissions = False
```

If you still want some of your views to have permission checks you can enable them via a decorator:

`cms.utils.decorators.cms_perms`

Here is a simple example:

```python
from cms.utils.decorators import cms_perms


@cms_perms
def my_view(request, **kw):
    ...
```

If you have your own permission check in your app, or just don't want to wrap some nested apps with CMS permission decorator, then use `exclude_permissions` property of apphook:

```python
class SampleApp(CMSApp):
    name = _("Sample App")
    urls = ["project.sampleapp.urls"]
    permissions = True
    exclude_permissions = ["some_nested_app"]
```

For example, django-oscar apphook integration needs to be used with exclude permissions of dashboard app, because it use customizable access function. So, your apphook in this case will looks like this:

```python
class OscarApp(CMSApp):
    name = _("Oscar")
    urls = [
        patterns('', *application.urls[0])
    ]
    exclude_permissions = ['dashboard']
```

Automatically restart server on apphook changes

As mentioned above, whenever you add or remove an apphook, change the slug of a page containing an apphook or the slug if a page which has a descendant with an apphook, you have to restart your server to re-load the URL caches. To allow you to automate this process, the django CMS provides a signal `cms.signals.urls_need_reloading` which you can listen on to detect when your server needs restarting. When you run `manage.py runserver` a restart should not be needed.

> **Warning:** This signal does not actually do anything. To get automated server restarting you need to implement logic in your project that gets executed whenever this signal is fired. Because there are many ways of deploying Django applications, there is no way we can provide a generic solution for this problem that will always work.

> **Warning:** The signal is fired **after** a request. If you change something via API you need a request for the signal to fire.

5.2.5 Working with templates

Application can reuse cms templates by mixing cms templatetags and normal django templating language.

static_placeholder

Plain *placeholder* cannot be used in templates used by external applications, use *static_placeholder* instead.

CMS_TEMPLATE

New in version 3.0.

CMS_TEMPLATE is a context variable available in the context; it contains the template path for CMS pages and application using apphooks, and the default template (i.e.: the first template in *CMS_TEMPLATES*) for non-CMS managed urls.

This is mostly useful to use it in the extends templatetag in the application templates to get the current page template.

Example: cms template

```
{% load cms_tags %}
<html>
    <body>
    {% cms_toolbar %}
    {% block main %}
    {% placeholder "main" %}
    {% endblock main %}
    </body>
</html>
```

Example: application template

```
{% extends CMS_TEMPLATE %}
{% load cms_tags %}
{% block main %}
{% for item in object_list %}
    {{ item }}
{% endfor %}
{% static_placeholder "sidebar" %}
{% endblock main %}
```

CMS_TEMPLATE memorizes the path of the cms template so the application template can dynamically import it.

render_model

New in version 3.0.

render_model allows to edit the django models from the frontend by reusing the django CMS frontend editor.

5.2.6 Extending the page & title models

New in version 3.0.

You can extend the page and title models with your own fields (e.g. adding an icon for every page) by using the extension models: `cms.extensions.PageExtension` and `cms.extensions.TitleExtension`, respectively.

What's the difference?

The difference between a page extension and a title extension is related to the difference between the Page and Title models. Titles support pages by providing a storage mechanism, among other things, for language-specific properties of Pages. So, if you find that you need to extend the page model in a language-specific manner, for example, if you need to create language- specific keywords for each language of your pages, then you may need to use a TitleExtension. If the extension you'd like to create is the same for all of the different languages of the page, then you may be fine using a PageExtension.

How To

To add a field to the page model, create a class that inherits from `cms.extensions.PageExtension`. Make sure to import the `cms.extensions.PageExtension` model. Your class should live in one of your apps' `models.py` (or module). Since `PageExtension` (and `TitleExtension`) inherit from `django.db.models.Model`, you are free to add any field you want but make sure you don't use a unique constraint on any of your added fields because uniqueness prevents the copy mechanism of the extension from working correctly. This means that you can't use one-to-one relations on the extension model. Finally, you'll need to register the model with using `extension_pool`.

Here's a simple example which adds an `icon` field to the page:

```python
from django.db import models

from cms.extensions import PageExtension
from cms.extensions.extension_pool import extension_pool


class IconExtension(PageExtension):
    image = models.ImageField(upload_to='icons')

extension_pool.register(IconExtension)
```

Hooking the extension to the admin site

To make your extension editable, you must first create an admin class that subclasses `cms.extensions.PageExtensionAdmin`. This admin handles page permissions. If you want to use your own admin class, make sure to exclude the live versions of the extensions by using `filter(extended_page__publisher_is_draft=True)` on the queryset.

Continuing with the example model above, here's a simple corresponding `PageExtensionAdmin` class:

```python
from django.contrib import admin
from cms.extensions import PageExtensionAdmin

from .models import IconExtension


class IconExtensionAdmin(PageExtensionAdmin):
    pass

admin.site.register(IconExtension, IconExtensionAdmin)
```

Since PageExtensionAdmin inherits from ModelAdmin, you'll be able to use the normal set of Django ModelAdmin properties, as appropriate to your circumstance.

Once you've registered your admin class, a new model will appear in the top- level admin list.

Note that the field that holds the relationship between the extension and a CMS Page is non-editable, so it will not appear in the admin views. This, unfortunately, leaves the operator without a means of "attaching" the page extension to the correct pages. The way to address this is to use a CMSToolbar. Note that creating the admin hook is still required, because it creates the add and change admin forms that are required for the next step.

Adding a Toolbar Menu Item for your Page extension

You'll also want to make your model editable from the cms toolbar in order to associate each instance of the extension model with a page. (Page isn't an editable attribute in the default admin interface.). To add toolbar items for your extension create a file named `cms_toolbar.py` in one of your apps, and add the relevant menu entries for the extension on each page.

Simplified toolbar API

New in version 3.0.6.

Since 3.0.6 a simplified toolbar API is available to handle the more common cases:

```python
from cms.toolbar_pool import toolbar_pool
from cms.extensions.toolbar import ExtensionToolbar
from django.utils.translation import ugettext_lazy as _
from .models import IconExtension


@toolbar_pool.register
class IconExtensionToolbar(ExtensionToolbar):
    # defineds the model for the current toolbar
    model = IconExtension

    def populate(self):
        # setup the extension toolbar with permissions and sanity checks
        current_page_menu = self._setup_extension_toolbar()
        # if it's all ok
        if current_page_menu:
            # retrieves the instance of the current extension (if any) and the toolbar item url
            page_extension, url = self.get_page_extension_admin()
            if url:
                # adds a toolbar item
                current_page_menu.add_modal_item(_('Page Icon'), url=url,
                    disabled=not self.toolbar.edit_mode)
```

Similarly for title extensions:

```python
from cms.extensions.toolbar import ExtensionToolbar
from django.utils.translation import ugettext_lazy as _
from .models import TitleIconExtension

@toolbar_pool.register
class TitleIconExtensionToolbar(ExtensionToolbar):
    # setup the extension toolbar with permissions and sanity checks
    model = TitleIconExtension

    def populate(self):
        # setup the extension toolbar with permissions and sanity checks
        current_page_menu = self._setup_extension_toolbar()
        # if it's all ok
        if current_page_menu and self.toolbar.edit_mode:
```

```
            # create a sub menu
            position = 0
            sub_menu = self._get_sub_menu(current_page_menu, 'submenu_label', 'Submenu', position)
            # retrieves the instances of the current title extension (if any) and the toolbar item
            urls = self.get_title_extension_admin()
            # cycle through the title list
            for title_extension, url in urls:
                # adds toolbar items
                sub_menu.add_modal_item('icon for title %s' % self._page().get_title(),
                                        url=url, disabled=not self.toolbar.edit_mode)
```

For details see the *reference*

Complete toolbar API

If you need complete control over the layout of your extension toolbar items you can still use the low-level API to
edit the toolbar according to your needs:

```python
from cms.api import get_page_draft
from cms.toolbar_pool import toolbar_pool
from cms.toolbar_base import CMSToolbar
from cms.utils import get_cms_setting
from cms.utils.permissions import has_page_change_permission
from django.core.urlresolvers import reverse, NoReverseMatch
from django.utils.translation import ugettext_lazy as _
from .models import IconExtension


@toolbar_pool.register
class IconExtensionToolbar(CMSToolbar):
    def populate(self):
        # always use draft if we have a page
        self.page = get_page_draft(self.request.current_page)

        if not self.page:
            # Nothing to do
            return

        # check global permissions if CMS_PERMISSION is active
        if get_cms_setting('PERMISSION'):
            has_global_current_page_change_permission = has_page_change_permission(self.request)
        else:
            has_global_current_page_change_permission = False
            # check if user has page edit permission
        can_change = self.request.current_page and self.request.current_page.has_change_permission
        if has_global_current_page_change_permission or can_change:
            try:
                icon_extension = IconExtension.objects.get(extended_object_id=self.page.id)
            except IconExtension.DoesNotExist:
                icon_extension = None
            try:
                if icon_extension:
                    url = reverse('admin:myapp_iconextension_change', args=(icon_extension.pk,))
                else:
                    url = reverse('admin:myapp_iconextension_add') + '?extended_object=%s' % self
            except NoReverseMatch:
                # not in urls
                pass
            else:
                not_edit_mode = not self.toolbar.edit_mode
                current_page_menu = self.toolbar.get_or_create_menu('page')
                current_page_menu.add_modal_item(_('Page Icon'), url=url, disabled=not_edit_mode)
```

Now when the operator invokes "Edit this page..." from the toolbar, there will be an additional menu item `Page Icon ...` (in this case), which can be used to open a modal dialog where the operator can affect the new `icon` field.

Note that when the extension is saved, the corresponding page is marked as having unpublished changes. To see the new extension values make sure to publish the page.

Using extensions with menus

If you want the extension to show up in the menu (e.g. if you had created an extension that added an icon to the page) use menu modifiers. Every `node.id` corresponds to their related `page.id`. `Page.objects.get(pk=node.id)` is the way to get the page object. Every page extension has a one-to-one relationship with the page so you can access it by using the reverse relation, e.g. `extension = page.yourextensionlowercased`. Now you can hook this extension by storing it on the node: `node.extension = extension`. In the menu template you can access your icon on the child object: `child.extension.icon`.

Using extensions in templates

To access a page extension in page templates you can simply access the approriate related_name field that is now available on the Page object.

As per the normal related_name naming mechanism, the appropriate field to access is the same as your Page-Extension model name, but lowercased. Assuming your Page Extension model class is `IconExtension`, the relationship to the page extension model will be available on `page.iconextension`. From there you can access the extra fields you defined in your extension, so you can use something like:

```
{% load staticfiles %}

{# rest of template omitted ... #}

{% if request.current_page.iconextension %}
    <img src="{% static request.current_page.iconextension.image.url %}">
{% endif %}
```

Where `request.current_page` is the normal way to access the current page that is rendering the template.

It is important to remember that unless the operator has already assigned a page extension to every page, a page may not have the iconextension relationship available, hence the use of the `{% if ... %}...{% endif %}` above.

Handling relations

If your PageExtension or TitleExtension includes a ForeignKey *from* another model or includes a ManyToMany field, you should also override the method `copy_relations(self, oldinstance, language)` so that these fields are copied appropriately when the CMS makes a copy of your extension to support versioning, etc.

Here's an example that uses a *ManyToMany'* field:

```python
from django.db import models
from cms.extensions import PageExtension
from cms.extensions.extension_pool import extension_pool


class MyPageExtension(PageExtension):

    page_categories = models.ManyToMany('categories.Category', blank=True, null=True)

    def copy_relations(self, oldinstance, language):
        for page_category in oldinstance.page_categories.all():
```

```
            page_category.pk = None
            page_category.mypageextension = self
            page_category.save()

extension_pool.register(MyPageExtension)
```

Simplified Toolbar API

The simplified Toolbar API works by deriving your toolbar class from `ExtensionToolbar` which provides the following API:

- `cms.extensions.toolbar.ExtensionToolbar._setup_extension_toolbar()`: this must be called first to setup the environment and do the permission checking;

- `cms.extensions.toolbar.ExtensionToolbar.get_page_extension_admin()`: for page extensions, retrieves the correct admin url for the related toolbar item; returns the extension instance (or *None* if not exists) and the admin url for the toolbar item;

- `cms.extensions.toolbar.ExtensionToolbar.get_title_extension_admin()`: for title extensions, retrieves the correct admin url for the related toolbar item; returns a list of the extension instances (or *None* if not exists) and the admin urls for each title of the current page;

5.2.7 Extending the Toolbar

New in version 3.0.

You can add and remove items to the toolbar. This allows you to integrate your application in the frontend editing mode of django CMS and provide your users with a streamlined editing experience.

For the toolbar API reference, please refer to *cms.toolbar*.

Registering

There are two ways to control what gets shown in the toolbar.

One is the *CMS_TOOLBARS*. This gives you full control over which classes are loaded, but requires that you specify them all manually.

The other is to provide `cms_toolbar.py` files in your apps, which will be automatically loaded as long *CMS_TOOLBARS* is not set (or set to `None`).

If you use the automated way, your `cms_toolbar.py` file should contain classes that extend from `cms.toolbar_base.CMSToolbar` and are registered using `cms.toolbar_pool.toolbar_pool.register()`. The register function can be used as a decorator.

These have four attributes: * toolbar (the toolbar object) * request (the current request) * is_current_app (a flag indicating whether the current request is handled by the same app as the function is in) * app_path (the name of the app used for the current request)

These classes must implement a `populate` or `post_template_populate` function. An optional `request_hook` function is available for overwrite as well.

- The populate functions will only be called if the current user is a staff user.

- The `populate` function will be called before the template and plugins are rendered.

- The `post_template_populate` function will be called after the template is rendered.

- The `request_hook` function is called before the view and may return a response. This way you would be able to issue redirects from a toolbar if needed

---

These classes can define an optional `supported_apps` attribute, specifying which applications the toolbar will work with. This is useful when the toolbar is defined in a different application from the views it's related to. to define the toolbar is local to (i.e.: for which the `supported_apps` is a tuple of application dotted paths (e.g: `supported_apps = ('whatever.path.app', 'another.path.app')`.

A simple example, registering a class that does nothing:

```python
from cms.toolbar_pool import toolbar_pool
from cms.toolbar_base import CMSToolbar


@toolbar_pool.register
class NoopModifier(CMSToolbar):

    def populate(self):
        pass

    def post_template_populate(self):
        pass

    def request_hook(self):
        pass
```

> **Warning:** As the toolbar passed to `post_template_populate` has been already populated with items from other applications, it might contains different items that when processed by `populate`.

> **Tip:** You can change the toolbar or add items inside a plugin render method (context['request'].toolbar) or inside a view (request.toolbar)

Adding items

Items can be added through the various *APIs* exposed by the toolbar and its items.

To add a *`cms.toolbar.items.Menu`* to the toolbar, use *`cms.toolbar.toolbar.CMSToolbar.get_or_create_me`* which will either add a menu if it doesn't exist, or create it.

Then, to add a link to your changelist that will open in the sideframe, use the *`cms.toolbar.items.ToolbarMixin.add_sideframe_item()`* method on the menu object returned.

When adding items, all arguments other than the name or identifier should be given as **keyword arguments**. This will help ensure that your custom toolbar items survive upgrades.

Following our Extending the Toolbar, let's add the poll app to the toolbar:

```python
from django.core.urlresolvers import reverse
from django.utils.translation import ugettext_lazy as _
from cms.toolbar_pool import toolbar_pool
from cms.toolbar_base import CMSToolbar


@toolbar_pool.register
class PollToolbar(CMSToolbar):

    def populate(self):
        if self.is_current_app:
            menu = self.toolbar.get_or_create_menu('poll-app', _('Polls'))
            url = reverse('admin:polls_poll_changelist')
            menu.add_sideframe_item(_('Poll overview'), url=url)
```

However, there's already a menu added by the CMS which provides access to various admin views, so you might want to add your menu as a sub menu there. To do this, you can use positional insertion coupled with the fact that *`cms.toolbar.toolbar.CMSToolbar.get_or_create_menu()`* will return already existing menus:

```
from django.core.urlresolvers import reverse
from django.utils.translation import ugettext_lazy as _
from cms.toolbar_pool import toolbar_pool
from cms.toolbar.items import Break
from cms.cms_toolbar import ADMIN_MENU_IDENTIFIER, ADMINISTRATION_BREAK
from cms.toolbar_base import CMSToolbar


@toolbar_pool.register
class PollToolbar(CMSToolbar):

    def populate(self):
        admin_menu = self.toolbar.get_or_create_menu(ADMIN_MENU_IDENTIFIER, _('Site'))
        position = admin_menu.find_first(Break, identifier=ADMINISTRATION_BREAK)
        menu = admin_menu.get_or_create_menu('poll-menu', _('Polls'), position=position)
        url = reverse('admin:polls_poll_changelist')
        menu.add_sideframe_item(_('Poll overview'), url=url)
        admin_menu.add_break('poll-break', position=menu)
```

If you wish to simply detect the presence of a menu without actually creating it, you can use
`cms.toolbar.toolbar.CMSToolbar.get_menu()`, which will return the menu if it is present, or, if
not, will return *None*.

Modifying an existing toolbar

If you need to modify an existing toolbar (say to change the `supported_apps` attribute) you can do this by
extending the original one, and modifying the appropriate attribute.

If *CMS_TOOLBARS* is used to register the toolbars, add your own toolbar instead of the original one, otherwise
unregister the original and register your own:

```
from cms.toolbar_pool import toolbar_pool
from third.party.app.cms.toolbar_base import FooToolbar


@toolbar_pool.register
class BarToolbar(FooToolbar):
    supported_apps = ('third.party.app', 'your.app')


toolbar_pool.unregister(FooToolbar)
```

Adding Items Alphabetically

Sometimes it is desirable to add sub-menus from different applications alphabetically. This can be challenging
due to the non-obvious manner in which your apps will be loaded into Django and is further complicated when
you add new applications over time.

To aide developers, django-cms exposes a `cms.toolbar.items.ToolbarMixin.get_alphabetical_insert_posit`
method, which, if used consistently can produce alphabetized sub-menus, even when they come from multiple
applications.

An example is shown here for an 'Offices' app, which allows handy access to certain admin functions for managing
office locations in a project:

```
from django.core.urlresolvers import reverse
from django.utils.translation import ugettext_lazy as _
from cms.toolbar_base import CMSToolbar
from cms.toolbar_pool import toolbar_pool
from cms.toolbar.items import Break, SubMenu
from cms.cms_toolbar import ADMIN_MENU_IDENTIFIER, ADMINISTRATION_BREAK


@toolbar_pool.register
class OfficesToolbar(CMSToolbar):
```

```python
def populate(self):
    #
    # 'Apps' is the spot on the existing djang-cms toolbar admin_menu
    # 'where we'll insert all of our applications' menus.
    #
    admin_menu = self.toolbar.get_or_create_menu(
        ADMIN_MENU_IDENTIFIER, _('Apps')
    )

    #
    # Let's check to see where we would insert an 'Offices' menu in the
    # admin_menu.
    #
    position = admin_menu.get_alphabetical_insert_position(
        _('Offices'),
        SubMenu
    )

    #
    # If zero was returned, then we know we're the first of our
    # applications' menus to be inserted into the admin_menu, so, here
    # we'll compute that we need to go after the first
    # ADMINISTRATION_BREAK and, we'll insert our own break after our
    # section.
    #
    if not position:
        # OK, use the ADMINISTRATION_BREAK location + 1
        position = admin_menu.find_first(
            Break,
            identifier=ADMINISTRATION_BREAK
        ) + 1
        # Insert our own menu-break, at this new position. We'll insert
        # all subsequent menus before this, so it will ultimately come
        # after all of our applications' menus.
        admin_menu.add_break('custom-break', position=position)

    # OK, create our office menu here.
    office_menu = admin_menu.get_or_create_menu(
        'offices-menu',
        _('Offices ...'),
        position=position
    )

    # Let's add some sub-menus to our office menu that help our users
    # manage office-related things.

    # Take the user to the admin-listing for offices...
    url = reverse('admin:offices_office_changelist')
    office_menu.add_sideframe_item(_('Offices List'), url=url)

    # Display a modal dialogue for creating a new office...
    url = reverse('admin:offices_office_add')
    office_menu.add_modal_item(_('Add New Office'), url=url)

    # Add a break in the submenus
    office_menu.add_break()

    # More submenus...
    url = reverse('admin:offices_state_changelist')
    office_menu.add_sideframe_item(_('States List'), url=url)

    url = reverse('admin:offices_state_add')
```

```
        office_menu.add_modal_item(_('Add New State'), url=url)
```

Here is the resulting toolbar (with a few other menus sorted alphabetically beside it)
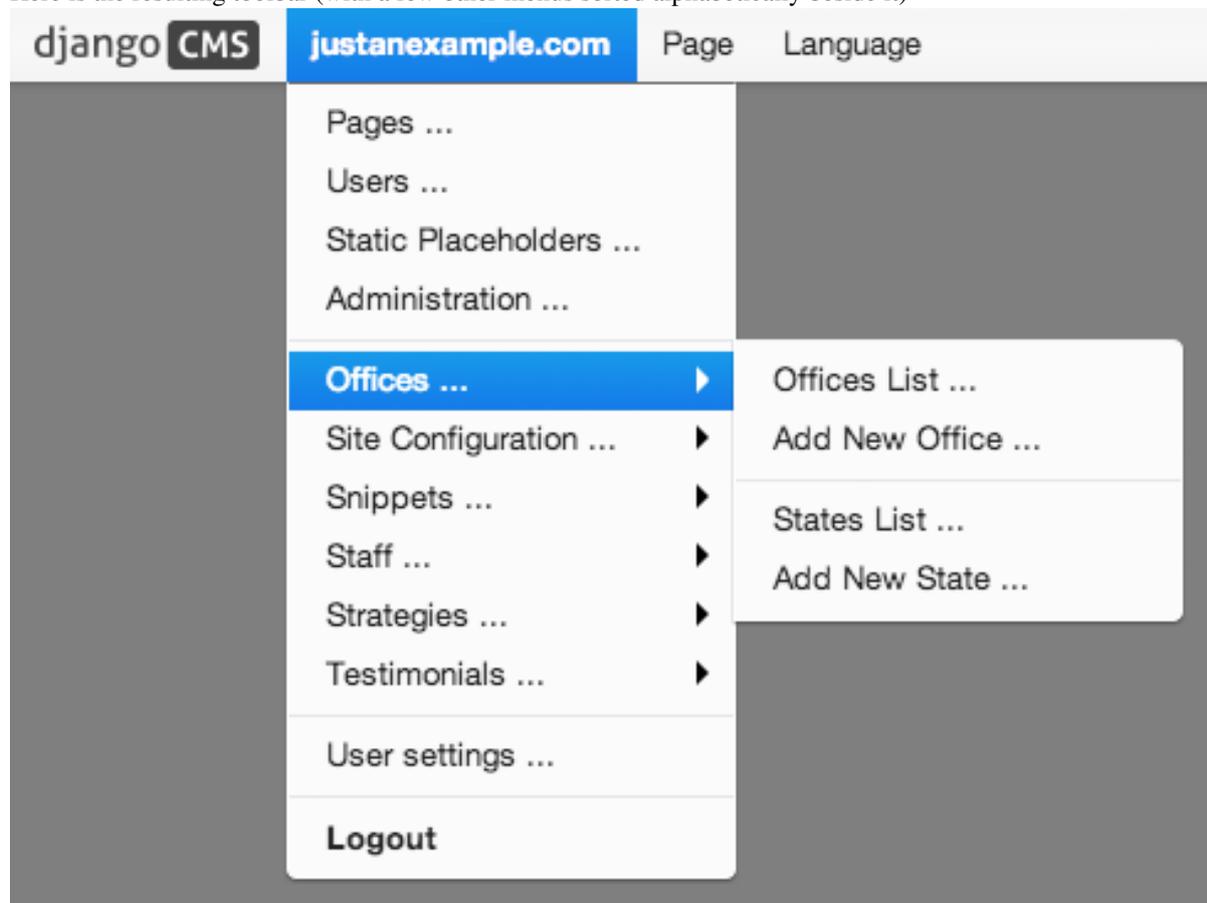
Adding items through views

Another way to add items to the toolbar is through our own views (`polls/views.py`). This method can be useful if you need to access certain variables, in our case e.g. the selected poll and its sub-methods:

```python
from django.core.urlresolvers import reverse
from django.shortcuts import get_object_or_404, render
from django.utils.translation import ugettext_lazy as _

from polls.models import Poll


def detail(request, poll_id):
    poll = get_object_or_404(Poll, pk=poll_id)
    menu = request.toolbar.get_or_create_menu('polls-app', _('Polls'))
    menu.add_modal_item(_('Change this Poll'), url=reverse('admin:polls_poll_change', args=[poll_
    menu.add_sideframe_item(_('Show History of this Poll'), url=reverse('admin:polls_poll_history
    menu.add_sideframe_item(_('Delete this Poll'), url=reverse('admin:polls_poll_delete', args=[po

    return render(request, 'polls/detail.html', {'poll': poll})
```

Detecting url changes Sometimes toolbar entries allow that you change the url of the current object displayed in the website. For example you are inside a blog entry and the toolbar allows to edit the blog slug or url. The toolbar will watch the `django.contrib.admin.models.LogEntry` model and detect if you create or edit

an object in the admin via modal or sideframe view. After the modal or sideframe closes it will redirect to the new url of the object.

To set this behavior manually you can set the `request.toolbar.set_object()` function on which you can set the current object.

Example:

```python
def detail(request, poll_id):
    poll = get_object_or_404(Poll, pk=poll_id)
    if hasattr(request, 'toolbar'):
        request.toolbar.set_object(poll)
    return render(request, 'polls/detail.html', {'poll': poll})
```

If you want to watch for object creation or editing of models and redirect after they have been added or changed add a `watch_models` attribute to your toolbar.

Example:

```python
class PollToolbar(CMSToolbar):

    watch_models = [Poll]

    def populate(self):
        ...
```

After you add this every change to an instance of `Poll` via sideframe or modal window will trigger a redirect to the URL of the poll instance that was edited, according to the toolbar status: if in *draft* mode the `get_draft_url()` is returned (or `get_absolute_url()` if the former does not exists), if in *live* mode and the method exists `get_public_url()` is returned;

Frontend

The toolbar adds a class `cms-ready` to the **html** tag when ready. Additionally we add `cms-toolbar-expanded` when the toolbar is visible (expanded).

The toolbar also fires a JavaScript event called **cms-ready** on the document. You can listen to this event using jQuery:

```
CMS.$(document).on('cms-ready', function () { ...  });
```

5.2.8 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.

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.

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.

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.

5.2.9 Testing Your Extensions

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/urls.py` with the following code:

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

urlpatterns = [
    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:

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


class MyappTests(CMSTestCase):

    @override_settings(ROOT_URLCONF='myapp.tests.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:

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


@override_settings(ROOT_URLCONF='myapp.tests.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.

Testing Plugins

To test plugins, you need to assign them to a placeholder. Depending on at what level you want to test your plugin,
you can either check the HTML generated by it or the context provided to its template:

```python
from django.test import TestCase

from cms.api import add_plugin
from cms.models import Placeholder

from myapp.cms_plugins import MyPlugin
from myapp.models import MyappPlugin


class MypluginTests(TestCase):
    def test_plugin_context(self):
        placeholder = Placeholder.objects.create(slot='test')
        model_instance = add_plugin(
            placeholder,
            MyPlugin,
            'en',
        )
        plugin_instance = model_instance.get_plugin_class_instance()
        context = plugin_instance.render({}, model_instance, None)
        self.assertIn('key', context)
        self.assertEqual(context['key'], 'value')

    def test_plugin_html(self):
        placeholder = Placeholder.objects.create(slot='test')
        model_instance = add_plugin(
            placeholder,
            MyPlugin,
            'en',
        )
        html = model_instance.render_plugin({})
        self.assertEqual(html, '<strong>Test</strong>')
```

5.2.10 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 the same frontend
editing as the CMS pages.

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`.

Quickstart

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

```python
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` has one required parameter, a string `slotname`.

The `slotname` is used in templates, to determine where the placeholder's plugins should appear in the page, and in the placeholder configuration *CMS_PLACEHOLDER_CONF*, which determines which plugins may be inserted into this placeholder.

You can also use a callable for the `slotname`, as in:

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


def my_placeholder_slotname(instance):
    return 'placeholder_name'


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

> **Warning:** For security reasons the related_name for a `PlaceholderField` may not be suppressed using `'+'`; this allows the cms to check permissions properly. Attempting to do so will raise a `ValueError`.

> **Note:** If you add a PlaceholderField to an existing model, you'll be able to see the placeholder in the frontend editor only after saving the relevant instance.

Admin Integration

Changed in version 3.0.

Your model with `PlaceholderFields` can still be edited in the admin. However, any PlaceholderFields in it will **only be available for editing from the frontend**. `PlaceholderFields` **must** not be present in any `fieldsets`, `fields`, `form` or other modeladmin field's definition attribute.

To provide admin support for a model with a `PlaceholderField` in your application's admin, you need to use the mixin `PlaceholderAdminMixin` along with the `ModelAdmin`. Note that the `PlaceholderAdminMixin` **must** precede the `ModelAdmin` in the class definition:

```
from django.contrib import admin
from cms.admin.placeholderadmin import PlaceholderAdminMixin
from myapp.models import MyModel


class MyModelAdmin(PlaceholderAdminMixin, admin.ModelAdmin):
    pass

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

I18N Placeholders

Out of the box `PlaceholderAdminMixin` supports multiple languages and will display language tabs. If you extend your model admin class derived from `PlaceholderAdminMixin` and overwrite `change_form_template` have a look at

admin/placeholders/placeholder/change_form.html to see how to display the language tabs.

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

```python
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 PlaceholderAdminMixin when registering your model with the admin site:

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


class MultilingualModelAdmin(TranslatableAdmin, PlaceholderAdminMixin, admin.ModelAdmin):
    pass

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

Templates

To render the placeholder in a template you use the *render_placeholder* tag from the cms_tags template tag library:

```
{% load cms_tags %}

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

The *render_placeholder* tag takes the following parameters:

- PlaceholderField instance

- width parameter for context sensitive plugins (optional)

- language keyword plus language-code string to render content in the specified language (optional)

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:

```python
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 cms_tags %}

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

Adding content to a placeholder

Changed in version 3.0.

Placeholders can be edited from the frontend by visiting the page displaying your model (where you put the `render_placeholder` tag), then appending `?edit` to the page's URL.

This will make the frontend editor top banner appear (and if necessary will require you to login).

Once in frontend editing mode, the interface for your application's `PlaceholderFields` will work in much the same way as it does for CMS Pages, with a switch for Structure and Content modes and so on.

There is no automatic draft/live functionality for general Django models, so content is updated instantly whenever you add/edit them.

Options

If you need to change `?edit` to a custom string (say, `?admin_on`) you may set `CMS_TOOLBAR_URL__EDIT_ON` variable in your `settings.py` to `"admin_on"`.

You may also change other URLs with similar settings:

- `?edit_off` (CMS_TOOLBAR_URL__EDIT_OFF)
- `?build` (CMS_TOOLBAR_URL__BUILD)
- `?toolbar_off` (CMS_TOOLBAR_URL__DISABLE)

When changing these settings, please be careful because you might inadvertently replace reserved strings in system (such as `?page`). We recommended you use safely unique strings for this option (such as `secret_admin` or `company_name`).

Permissions

To be able to edit a placeholder user must be a `staff` member and needs either edit permissions on the model that contains the `PlaceholderField`, or permissions for that specific instance of that model.

Model permissions are usually added through the default Django auth application and its admin interface. Object-level permission can be handled by writing a custom Auth Backend as described in django docs

For example, if there is a `UserProfile` model that contains a `PlaceholderField` then the custom backend can refer to a `has_perm` method (on the model) that grants all rights to current user only based on the user's `UserProfile` object:

```python
def has_perm(self, user_obj, perm, obj=None):
    if not user_obj.is_staff:
        return False
    if isinstance(obj, UserProfile):
        if user_obj.get_profile()==obj:
            return True
    return False
```

5.2.11 Caching

Setup

To setup caching configure a caching backend in django.

Details for caching can be found here: https://docs.djangoproject.com/en/dev/topics/cache/

In your middleware settings be sure to add `django.middleware.cache.UpdateCacheMiddleware` at the first and `django.middleware.cache.FetchFromCacheMiddleware` at the last position:

```
MIDDLEWARE_CLASSES=[
        'django.middleware.cache.UpdateCacheMiddleware',
        ...
        'cms.middleware.language.LanguageCookieMiddleware',
        'cms.middleware.user.CurrentUserMiddleware',
        'cms.middleware.page.CurrentPageMiddleware',
        'cms.middleware.toolbar.ToolbarMiddleware',
        'django.middleware.cache.FetchFromCacheMiddleware',
    ],
```

Plugins

New in version 3.0.

Normally all plugins will be cached. If you have a plugin that is dynamic based on the current user or other dynamic properties of the request set the `cache=False` attribute on the plugin class:

```python
class MyPlugin(CMSPluginBase):
    name = _("MyPlugin")
    cache = False
```

> **Warning:** If you disable a plugin cache be sure to restart the server and clear the cache afterwards.

Content Cache Duration

Default: 60

This can be changed in *CMS_CACHE_DURATIONS*

Settings

Caching is set default to true. Have a look at the following settings to enable/disable various caching behaviors:

- *CMS_PAGE_CACHE*
- *CMS_PLACEHOLDER_CACHE*
- *CMS_PLUGIN_CACHE*

5.2.12 Frontend editing for Page and Django models

New in version 3.0.

As well as `PlaceholderFields`, 'ordinary' Django model fields (both on CMS Pages and your own Django models) can also be edited through django CMS's frontend editing interface. This is very convenient for the user because it saves having to switch between frontend and admin views.

Using this interface, model instance values that can be edited show the "Double-click to edit" hint on hover. Double-clicking opens a pop-up window containing the changeform for that model.

> **Warning:** By default and for consistency with previous releases, templatetags used by this feature mark as safe the content of the rendered model attribute. This may be a security risk if used on fields which may hold non-trusted content. Be aware, and use the templatetags accordingly. To change this behaviour, set the setting: `CMS_UNESCAPED_RENDER_MODEL_TAGS` to False.

> **Warning:** This feature is only partially compatible with django-hvad: using `render_model` with hvad-translated fields (say {% render_model object 'translated_field' %} return error if the hvad-enabled object does not exists in the current language. As a workaround `render_model_icon` can be used instead.

Templatetags

This feature relies on five templatetags sharing common code:

- *render_model*
- *render_model_icon*
- *render_model_add*
- *render_model_block*
- *render_model_add_block*

Look at the tag-specific page for a detailed reference; in the examples below `render_model` is assumed.

Page titles edit

For CMS pages you can edit the titles from the frontend; according to the attribute specified a overridable default field will be editable.

Main title:

```
{% render_model request.current_page "title" %}
```

Page title:

```
{% render_model request.current_page "page_title" %}
```

Menu title:

```
{% render_model request.current_page "menu_title" %}
```

All three titles:

```
{% render_model request.current_page "titles" %}
```

You can always customise the editable fields by providing the *edit_field* parameter:

```
{% render_model request.current_page "title" "page_title,menu_title" %}
```

Page menu edit

By using the special keyword `changelist` as edit field the frontend editing will show the page tree; a common pattern for this is to enable changes in the menu by wrapping the menu templatetags:

```
{% render_model_block request.current_page "changelist" %}
    <h3>Menu</h3>
    <ul>
        {% show_menu 1 100 0 1 "sidebar_submenu_root.html" %}
    </ul>
{% endrender_model_block %}
```

Will render to:

```html
<div class="cms_plugin cms_plugin-cms-page-changelist-1">
    <h3>Menu</h3>
    <ul>
        <li><a href="/">Home</a></li>
        <li><a href="/another">another</a></li>
        [...]
</div>
```

Editing 'ordinary' Django models

As noted above, your own Django models can also present their fields for editing in the frontend. This is achieved by using the `FrontendEditableAdminMixin` base class.

Note that this is only required for fields **other than** `PlaceholderFields`. `PlaceholderFields` are automatically made available for frontend editing.

Configure the model's admin class

Configure your admin class by adding the `FrontendEditableAdminMixin` mixin to it (see Django admin documentation for general Django admin information).

```python
from cms.admin.placeholderadmin import FrontendEditableAdminMixin
from django.contrib import admin


class MyModelAdmin(FrontendEditableAdminMixin, admin.ModelAdmin):
    ...
```

The ordering is important: as usual, **mixins must come first**.

Then set up the templates where you want to expose the model for editing, adding a `render_model` templatetag:

```django
{% load cms_tags %}

{% block content %}
<h1>{% render_model instance "some_attribute" %}</h1>
{% endblock content %}
```

See *templatetag reference* for arguments documentation.

Selected fields edit

Frontend editing is also possible for a set of fields.

Set up the admin   You need to add to your model admin a tuple of fields editable from the frontend admin:

```python
from cms.admin.placeholderadmin import FrontendEditableAdminMixin
from django.contrib import admin


class MyModelAdmin(FrontendEditableAdminMixin, admin.ModelAdmin):
    frontend_editable_fields = ("foo", "bar")
    ...
```

Set up the template   Then add comma separated list of fields (or just the name of one field) to the templatetag:

```
{% load cms_tags %}

{% block content %}
<h1>{% render_model instance "some_attribute" "some_field,other_field" %}</h1>
{% endblock content %}
```

Special attributes

The `attribute` argument of the templatetag is not required to be a model field, property or method can also be used as target; in case of a method, it will be called with request as argument.

Custom views

You can link any field to a custom view (not necessarily an admin view) to handle model-specific editing workflow.

The custom view can be passed either as a named url (`view_url` parameter) or as name of a method (or property) on the instance being edited (`view_method` parameter). In case you provide `view_method` it will be called whenever the templatetag is evaluated with `request` as parameter.

The custom view does not need to obey any specific interface; it will get `edit_fields` value as a `GET` parameter.

See *templatetag reference* for arguments documentation.

Example `view_url`:

```
{% load cms_tags %}

{% block content %}
<h1>{% render_model instance "some_attribute" "some_field,other_field" "" "admin:exampleapp_exampl
{% endblock content %}
```

Example `view_method`:

```
class MyModel(models.Model):
    char = models.CharField(max_length=10)

    def some_method(self, request):
        return "/some/url"


{% load cms_tags %}

{% block content %}
<h1>{% render_model instance "some_attribute" "some_field,other_field" "" "" "some_method" %}</h1
{% endblock content %}
```

Model changelist

By using the special keyword `changelist` as edit field the frontend editing will show the model changelist:

```
{% render_model instance "name" "changelist" %}
```

Will render to:

```
<div class="cms_plugin cms_plugin-myapp-mymodel-changelist-1">
    My Model Instance Name
</div>
```

Filters

If you need to apply filters to the output value of the templatetag, add quoted sequence of filters as in Django
`filter` templatetag:

```
{% load cms_tags %}

{% block content %}
<h1>{% render_model instance "attribute" "" "" "truncatechars:9" %}</h1>
{% endblock content %}
```

Context variable

The templatetag output can be saved in a context variable for later use, using the standard *as* syntax:

```
{% load cms_tags %}

{% block content %}
{% render_model instance "attribute" as variable %}

<h1>{{ variable }}</h1>

{% endblock content %}
```

5.2.13 Sitemap Guide

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

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

django.contrib.sitemaps

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

5.3 Key topics

This section explains and analyses some key concepts in django CMS. It's less concerned with explaining *how to
do things* than with helping you understand *how it works*.

5.3.1 How the menu system works

Basic concepts

Soft Roots

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 <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
```

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 classsses 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 able to access Page objects via NavigationNodes.

Menu system logic

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's menu.py, an**

            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.menu.CMSMenu`

                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_info, and marks the best match as `selected`

            loops over the Modifiers in self.modifiers calling each one's `modify(post_cut=False)()`. T

                `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 node (level = 0

                    `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 Modifier**

  · `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`

5.3.2 Publishing

Each published page in the CMS exists in as two `cms.Page` instances: **public** and **draft**.

Until it's published, only the **draft** version exists.

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 yet published, the page goes into a "pending" state. It will be automatically published once the parent page is published.

This enables you to edit an entire subsection of the website, publishing it only once all the work is complete.

Code and Pages

When handling `cms.Page` in code, you'll generally want to deal with draft instances.

Draft pages are the ones you interact with in the admin, and in draft mode in the CMS frontend. When a draft page is published, a public version is created and all titles, placeholders and plugins are copied to the public version.

The `cms.Page` model has a `publisher_is_draft` field, that's `True` for draft versions. Use a filter:

```
``publisher_is_draft=True``
```

to get hold of these draft `Page` instances.

5.3.3 Serving content in multiple languages

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 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.

5.3.4 Internationalisation

Multilingual URLs

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

Here's an example of *urls.py*:

```python
from django.conf import settings
from django.conf.urls import 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 = [
    url(r'^jsi18n/(?P<packages>\S+?)/$', 'django.views.i18n.javascript_catalog'),
]

urlpatterns += staticfiles_urlpatterns()

# note the django CMS URLs included via i18n_patterns
urlpatterns += i18n_patterns('',
    url(r'^admin/', include(admin.site.urls)),
    url(r'^', include('cms.urls')),
)
```

Language Cookie

If a user comes back to a previously visited page, the language will be same since his last visit.

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

More in-depth documentation about this is available at https://docs.djangoproject.com/en/dev/topics/i18n/translation/#how-django-discovers-language-preference

When visiting a page that is only English and French with a German browser, the language from LAN-GUAGE_CODE will be used. If this is English, but the visitor only speaks French, the visitor will have to switch the language. Visiting the same page now again after some time, will show it in English again, because the browser session which was used to store the language selection doesn't exist anymore. To prevent this issue, a middleware exists which stores the language selection in a cookie. Add the following middleware to *MIDDLE-WARE_CLASSES*:

> *cms.middleware.language.LanguageCookieMiddleware*

Language Chooser

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

Example:

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

If you are in an apphook and have a detail view of an object you can set an object to the toolbar in your view. The cms will call `get_absolute_url` in the corresponding language for the language chooser:

Example:

```
class AnswerView(DetailView):
    def get(self, *args, **kwargs):
        self.object = self.get_object()
        if hasattr(self.request, 'toolbar'):
            self.request.toolbar.set_object(self.object)
        response = super(AnswerView, self).get(*args, **kwargs)
        return response
```

With this you can more easily control what url will be returned on the language chooser.

Note: If you have a multilingual objects be sure that you return the right url if you don't have a translation for this language in `get_absolute_url`

---

page_language_url

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

Example:

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

hide_untranslated

If you add a default directive to your *CMS_LANGUAGES* with a `hide_untranslated` 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.

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.

5.3.5 Permissions

In django CMS you can set three 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

3. Mode permission which when left unset, restricts staff users to only editing, not adding new content

To enable features 1. and 2., `settings.py` requires:

```
CMS_PERMISSION = True
```

The third one is controlled by the "**Can use Structure mode**" Django permission.

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*)

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

Users (page) / User groups (page)

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

You would be able to create a 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*.

User groups (page) manages the same feature on the group level.

Page 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`.

Pages global permissions

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

Note: You always **must** set the sites managed py the global permissions, even if you only have one site.

Edit mode permission

Changed in version 3.1.

django CMS uses **Structure** and **Content** modes for different type of content editing; while the former allows full control over the plugins layout, positioning and to add new plugins to the page, the latter only allow editing existing plugins.

From version 3.1 the specific permission "**Can use Structure mode**" exists to permit access to Structure Mode. This allows defining a different level of permissions on the same content.

This permission also applies to `PlaceholderField` defined on models.

File Permissions

django CMS does not take care of and no responsibility for controlling access to files. Please make sure to use either a prebuilt solution (like django-filer) or to roll your own.

5.3.6 Some commonly-used plugins

> **Warning:** In version 3 of the CMS we removed all the plugins from the into separate repositories to continue their development there. you are upgrading from a previous version. Please refer to *Upgrading from previous versions*

These are the recommended plugins to use with django CMS.

> **Important:** See the note on *The INSTALLED_APPS setting* about ordering.

File

Available on GitHub (divio/djangocms-file) and on PyPi (djangocms-file).

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

Please install it using `pip` or similar and be sure you have the following in the `INSTALLED_APPS` setting in your project's `settings.py` file:

```
INSTALLED_APPS = (
    # ...
    'djangocms_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 filer CMS plugin and its `cmsplugin_filer_file` component instead.

> **Warning:** The `djangocms_file` 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.

Flash

Available on GitHub (divio/djangocms-flash) and on PyPi (djangocms-flash).

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

Please install it using `pip` or similar and be sure you have the following in the `INSTALLED_APPS` setting in your project's `settings.py` file:

```
INSTALLED_APPS = (
    # ...
    'djangocms_flash',
    # ...
)
```

GoogleMap

Available on GitHub (divio/djangocms-googlemap) and on PyPi (djangocms-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*.

Please install it using `pip` or similar and be sure you have the following in the `INSTALLED_APPS` setting in your project's `settings.py` file:

```
INSTALLED_APPS = (
    # ...
    'djangocms_googlemap',
    # ...
)
```

Picture

Available on GitHub (divio/djangocms-picture) and on PyPi (djangocms-picture).

Displays a picture in a page.

Please install it using `pip` or similar and be sure you have the following in the `INSTALLED_APPS` setting in your project's `settings.py` file:

```
INSTALLED_APPS = (
    # ...
    'djangocms_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 `djangocms_picture/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 filer CMS plugin and its `cmsplugin_filer_image` component instead.

Teaser

Available on GitHub (divio/djangocms-teaser) and on PyPi (djangocms-teaser).

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

Please install it using `pip` or similar and be sure you have the following in the `INSTALLED_APPS` settings in your project's `settings.py` file:

```
INSTALLED_APPS = (
    # ...
    'djangocms_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 filer CMS plugin and its `cmsplugin_filer_teaser` component instead.

Text

Consider using djangocms-text-ckeditor for displaying text. You may of course use your preferred editor; others are available.

Video

Available on GitHub (divio/djangocms-video) and on PyPi (djangocms-video).

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

Please install it using `pip` or similar and be sure you have the following in your project's `INSTALLED_APPS` setting:

```
INSTALLED_APPS = (
    # ...
    'djangocms_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 filer CMS plugin and its cmsplugin_filer_video component instead.

Twitter

We recommend one of the following plugins:

- https://github.com/nephila/djangocms_twitter

- https://github.com/changer/cmsplugin-twitter

Warning: These plugins are not currently compatible with Django 1.7.

Inherit

Available on GitHub (divio/djangocms-inherit) and on PyPi (djangocms-inherit).

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

Please install it using pip or similar and be sure you have the following in your project's INSTALLED_APPS setting:

```
INSTALLED_APPS = (
    # ...
    'djangocms_inherit',
    # ...
)
```

Warning: The inherit plugin **cannot** be used in non-cms placeholders.

5.3.7 Search and django CMS

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

5.4 Reference

Technical reference material.

5.4.1 Configuration

django CMS has a number of settings to configure its behaviour. These should be available in your `settings.py` file.

The `INSTALLED_APPS` setting

The ordering of items in `INSTALLED_APPS` matters. Entries for applications with plugins should come *after* `cms`.

Custom User Requirements

When using a custom user model (i.e. the AUTH_USER_MODEL Django setting), there are a few requirements that must be met.

DjangoCMS expects a user model with at minimum the following fields: email, password, is_active, is_staff, and is_superuser. Additionally, it should inherit from AbstractBaseUser and PermissionsMixin (or AbstractUser), and must define one field as the USERNAME_FIELD (see Django documentation for more details) and define a get_fullname() method.

The models must also be editable via django admin and have an admin class registered.

Additionally, the application in which the model is defined **must** be loaded before *cms* in *INSTALLED_APPS*.

Note: In most cases, it is better to create a UserProfile model with a one to one relationship to auth.User rather than creating a custom user model. Custom user models are only necessary if you intended to alter the default behavior of the User model, not simply extend it.

Additionally, if you do intend to use a custom user model, it is generally advisable to do so only at the beginning of a project, before the database is created.

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*.

Note: Alternatively you can use *CMS_TEMPLATES_DIR* to define a directory containing templates for django CMS.

> **Warning:** django CMS requires some special templates to function correctly. These are provided within `cms/templates/cms`. You are strongly advised not to use `cms` as a directory name for your own project templates.

Basic Customisation

CMS_TEMPLATE_INHERITANCE

default `True`

Enables the inheritance of templates from parent pages.

When enabled, pages' `Template` options will include a new default: *Inherit from the parent page* (unless the page is a root page).

CMS_TEMPLATES_DIR

default `None`

Instead of explicitly providing a set of templates via *CMS_TEMPLATES* a directory can be provided using this configuration.

CMS_TEMPLATES_DIR can be set to the (absolute) path of the templates directory, or set to a dictionary with *SITE_ID: template path* items:

```
CMS_TEMPLATES_DIR: {
    1: '/absolute/path/for/site/1/',
    2: '/absolute/path/for/site/2/',
}
```

The provided directory is scanned and all templates in it are loaded as templates for django CMS.

Template loaded and their names can be customized using the templates dir as a python module, by creating a `__init__.py` file in the templates directory. The file contains a single `TEMPLATES` dictionary with the list of templates as keys and template names as values:::

```python
# -*- coding: utf-8 -*-
from django.utils.translation import ugettext_lazy as _
TEMPLATES = {
    'col_two.html': _('Two columns'),
    'col_three.html': _('Three columns'),
}
```

Being a normal python file, templates labels can be passed through gettext for translation.

Note: As templates are still loaded by the Django template loader, the given directory **must** be reachable by the template loading system. Currently **filesystem** and **app_directory** loader schemas are tested and supported.

CMS_PLACEHOLDER_CONF

default `{}`

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

Example:

```
CMS_PLACEHOLDER_CONF = {
    'content': {
        'plugins': ['TextPlugin', 'PicturePlugin'],
        'text_only_plugins': ['LinkPlugin'],
        'extra_context': {"width":640},
        'name': gettext("Content"),
        'language_fallback': True,
        'default_plugins': [
            {
                'plugin_type': 'TextPlugin',
                'values': {
                    'body':'<p>Lorem ipsum dolor sit amet...</p>',
                },
            },
        ],
        'child_classes': {
            'TextPlugin': ['PicturePlugin', 'LinkPlugin'],
        },
        'parent_classes': {
            'LinkPlugin': ['TextPlugin'],
        },
    },
    'right-column': {
        "plugins": ['TeaserPlugin', 'LinkPlugin'],
        "extra_context": {"width": 280},
        'name': gettext("Right Column"),
        'limits': {
            'global': 2,
            'TeaserPlugin': 1,
            'LinkPlugin': 1,
        },
        'plugin_modules': {
            'LinkPlugin': 'Extra',
        },
        'plugin_labels': {
            'LinkPlugin': 'Add a link',
        },
    },
    'base.html content': {
        "plugins": ['TextPlugin', 'PicturePlugin', 'TeaserPlugin'],
        'inherit': 'content',
    },
}
```

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).

language_fallback When `True`, if the placeholder has no plugin for the current language it falls back to
the fallback languages as specified in *CMS_LANGUAGES*. Defaults to `True` since version 3.1.

default_plugins You can specify the list of default plugins which will be automagically added when the

placeholder will be created (or rendered). Each element of the list is a dictionary with following keys :

plugin_type The plugin type to add to the placeholder Example : 'TextPlugin'

values Dictionary to use for the plugin creation. It depends on the `plugin_type`. See the documentation of each plugin type to see which parameters are required and available. Example for a Textplugin: {'body':'<p>Lorem ipsum</p>'} Example for a LinkPlugin : {'name':'Django-CMS','url':'https://www.django-cms.org'}

children It is a list of dictionaries to configure default plugins to add as children for the current plugin (it must accepts children). Each dictionary accepts same args than dictionaries of `default_plugins` : `plugin_type`, `values`, `children` (yes, it is recursive).

Complete example of default_plugins usage:

```python
CMS_PLACEHOLDER_CONF = {
    'content': {
        'name' : _('Content'),
        'plugins': ['TextPlugin', 'LinkPlugin'],
        'default_plugins':[
            {
                'plugin_type':'TextPlugin',
                'values':{
                    'body':'<p>Great websites : %(_tag_child_1)s and %(_tag_child_2)s</p>'
                },
                'children':[
                    {
                        'plugin_type':'LinkPlugin',
                        'values':{
                            'name':'django',
                            'url':'https://www.djangoproject.com/'
                        },
                    },
                    {
                        'plugin_type':'LinkPlugin',
                        'values':{
                            'name':'django-cms',
                            'url':'https://www.django-cms.org'
                        },
                        # If using LinkPlugin from djangocms-link which
                        # accepts children, you could add some grandchildren :
                        # 'children' : [
                        #     ...
                        # ]
                    },
                ]
            },
        ]
    }
}
```

plugin_modules A dictionary of plugins and custom module names to group plugin in the toolbar UI.

plugin_labels A dictionary of plugins and custom labels to show in the toolbar UI.

child_classes A dictionary of plugin names with lists describing which plugins may be placed inside each plugin. If not supplied, all plugins can be selected.

parent_classes A dictionary of plugin names with lists describing which plugins may contain each plugin. If not supplied, all plugins can be selected.

require_parent A boolean indication whether that plugin requires another plugin as parent or not.

inherit Placeholder name or template name + placeholder name which inherit. In the example, the configuration for "base.html content" inherits from "content" and just overwrite the "plugins" setting to allow TeaserPlugin, thus you have not to duplicate your "content"'s configuration.

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 plugins' output *after* rendering. See Custom Plugins for more information.

CMS_APPHOOKS

default: `()`

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

By default, apphooks are auto-discovered in applications listed in all `INSTALLED_APPS`, by trying to import their `cms_app` module.

When `CMS_APPHOOKS` is set, auto-discovery is disabled.

Example:

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

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': True,
        'hide_untranslated': False,
    }
}
```

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

> **Warning:** Make sure you use **language codes** (*en-us*) and not **locale names** (*en_US*) here and in `LANGUAGES`. Use *check command* to check for correct syntax.

`CMS_LANGUAGES` has different options where you can define how different languages behave, with granular control.

On the first level you can set values for each `SITE_ID`. 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, but 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.

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 Determines whether this language is accessible in the frontend. You may want for example to keep a language private until your content has been fully translated.

type Boolean

default `True`

fallbacks A list of alternative languages, in order of preference, that are to be used if a page is not translated yet..

example `['de', 'fr']`

default `[]`

hide_untranslated Hide untranslated pages in menus

type Boolean

default `True`

redirect_on_fallback Determines behaviour when the preferred language is not available. If `True`, will redirect to the URL of the same page in the fallback language. If `False`, the content will be displayed in the fallback language, but there will be no redirect.

Note that this applies to the fallback behaviour of *pages*. Starting for 3.1 *placeholders* **will** default to the same behaviour. If you do not want a placeholder to follow a page's fallback behaviour, you must set its `language_fallback` to `False` in *CMS_PLACEHOLDER_CONF*, above.

type Boolean

default `True`

Unicode support for automated slugs

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, 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.

Example Add these to your project's settings:

```
CMS_UNIHANDECODE_HOST = '/static/unihandecode/'
CMS_UNIHANDECODE_VERSION = '1.0.0'
CMS_UNIHANDECODE_DECODERS = ['ja', 'zh', 'vn', 'kr', 'diacritic']
```

Add the library files from GitHub ojii/unihandecode.js tree/dist to your static folder:

```
project/
    static/
        unihandecode/
            unihandecode-1.0.0.core.min.js
            unihandecode-1.0.0.diacritic.min.js
            unihandecode-1.0.0.ja.min.js
            unihandecode-1.0.0.kr.min.js
            unihandecode-1.0.0.vn.min.js
            unihandecode-1.0.0.zh.min.js
```

More documentation is available on unihandecode.js' Read the Docs.

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_UNESCAPED_RENDER_MODEL_TAGS

default True

> **Warning:** In this version of django CMS, this setting has a default value of `True` to provide behavior consistent with previous releases. However, all developers are encouraged to set this value to `False` to help prevent a range of security vulnerabilities stemming from HTML, Javascript, and CSS Code Injection.

Important: This setting is deprecated and will be removed in a near-future release. Developers are encouraged to carefully consider the source of any content displayed by the render_model template tag and only add the optional template filter safe on model fields that are known to be cleansed of any malicious strings.

When this setting is removed, the render_model template tag will no longer automatically mark as "safe" their output. Any content that is intended to be displayed as rendered markup will require the safe filter applied when displaying with the render_model tag.

This setting affects how certain template tags display model-based content. In particular, the template tag: *render_model*.

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 need to ensure that the directory to which it points is writable by the user under which Django will be running.

URLs

Advanced Settings

CMS_PERMISSION

default False

When enabled, 3 new models are provided 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`

Determines whether pages without any view restrictions are public by default or staff only. Possible values are `all` and `staff`.

CMS_CACHE_DURATIONS

This dictionary carries the various cache duration settings.

`'content'`

default `60`

Cache expiration (in seconds) for *show_placeholder*, *page_url*, *placeholder* and *static_placeholder* 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_PAGE_CACHE

default `True`

Should the output of pages be cached? Takes the language, and timezone into account. Pages for logged in users are not cached. If the toolbar is visible the page is not cached as well.

CMS_PLACEHOLDER_CACHE

default `True`

Should the output of the various placeholder templatetags be cached? Takes the current language and timezone into account. If the toolbar is in edit mode or a plugin with `cache=False` is present the placeholders will not be cached.

CMS_PLUGIN_CACHE

default `True`

Default value of the `cache` attribute of plugins. Should plugins be cached by default if not set explicitly?

Warning: If you disable the plugin cache be sure to restart the server and clear the cache afterwards.

CMS_MAX_PAGE_HISTORY_REVERSIONS

default `15`

Configures how many undo steps are saved in the db excluding publish steps. In the page admin there is a `History` button to revert to previous version of a page. In the past, databases using django-reversion could grow huge. To help address this issue, only a limited number of *edit* revisions will now be saved.

This setting declares how many edit revisions are saved in the database. By default the newest 15 edit revisions are kept.

CMS_MAX_PAGE_PUBLISH_REVERSIONS

default `10`

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, databases using django-reversion could grow huge. To help address this issue, only a limited number of *published* revisions will now be saved.

This setting declares how many published revisions are saved in the database. By default the newest 10 published revisions are kept; all others are deleted when you publish a page.

If set to *0* all published revisions are kept, but you will need to ensure that the revision table does not grow excessively large.

CMS_TOOLBARS

default `None`

If defined, specifies the list of toolbar modifiers to be used to populate the toolbar as import paths. Otherwise, all available toolbars from both the CMS and the 3rd party apps will be loaded.

Example:

```
CMS_TOOLBARS = [
    # CMS Toolbars
    'cms.cms_toolbar.PlaceholderToolbar',
    'cms.cms_toolbar.BasicToolbar',
    'cms.cms_toolbar.PageToolbar',

    # 3rd Party Toolbar
    'aldryn_blog.cms_toolbar.BlogToolbar',
]
```

CMS_TOOLBAR_ANONYMOUS_ON

default `True`

This setting controls if anonymous users can see the CMS toolbar with a login form when `?edit` is appended to a URL. The default behaviour is to show the toolbar to anonymous users.

CMS_TOOLBAR_HIDE

default `False`

If `True`, the toolbar is hidden, even on CMS pages.

To determine the internal url of django cms, you need to assign CMS_APP_NAME to use when you include 'cms.urls'

Example:

```
urlpatterns += i18n_patterns('',
    url(r'^admin/', include(admin.site.urls)),
    url(r'^content/', include('cms.urls', app_name=settings.CMS_APP_NAME)),
)
```

CMS_DEFAULT_X_FRAME_OPTIONS

default `Page.X_FRAME_OPTIONS_INHERIT`

This setting is the default value for a Page's X Frame Options setting. This should be an integer preferably taken from the Page object e.g.

- X_FRAME_OPTIONS_INHERIT
- X_FRAME_OPTIONS_ALLOW
- X_FRAME_OPTIONS_SAMEORIGIN
- X_FRAME_OPTIONS_DENY

5.4.2 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 live in the `menus` application, which though tightly coupled to the `cms` application exists independently of it. Menus are usable by any application, not just by django CMS.

---

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 `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 should be rendered and at which level it should stop. If you have home as a root node (i.e. level 0) and don't want to display the root node(s), set `start_level` to 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):

```html
<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":

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

You can give it the same optional parameters as show_menu:

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

Unlike *show_menu*, however, soft roots will not affect the menu when using show_menu_below_id.

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 have its root. 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.

Fourth argument, template (default=menu/sub_menu.html), is the template used by the tag; if you want to use a different template you **must** supply default values for root_level and nephews.

Examples:

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

Rooted at level 0:

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

Or with a custom template:

```html
<ul>
    {% show_sub_menu 1 None 100 "myapp/submenu.html" %}
</ul>
```

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" %}
```

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

```
{% show_breadcrumb 0 "menu/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 %}
```

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.

```
{{ 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*. A page can be marked as a *soft root* in its 'Advanced Settings'.

Modifying & Extending the menu

Please refer to the Customising navigation menus documentation

5.4.3 Plugins

CMSPluginBase Attributes and Methods Reference

These are a list of attributes and methods that can (or should) be overridden on your Plugin definition.

Attributes

admin_preview   Default: `False`

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

allow_children   Default: `False`

Can this plugin have child plugins? Or can other plugins be placed inside this plugin? If set to `True` you are responsible to render the children in your plugin template.

Please use something like this or something similar:

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

Be sure to access `instance.child_plugin_instances` to get all children. They are pre-filled and ready to use. To finally render your child plugins use the `{% render_plugin %}` templatetag.

See also: *child_classes*, *parent_classes*, *require_parent*

cache   Default: *CMS_PLUGIN_CACHE*

Is this plugin cacheable? If your plugin displays content based on the user or request or other dynamic properties set this to False.

> **Warning:** If you disable a plugin cache be sure to restart the server and clear the cache afterwards.

change_form_template   Default: `admin/cms/page/plugin_change_form.html`

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

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"
```

See also: *frontend_edit_template*

child_classes   Default: `None`

A List of Plugin Class Names. If this is set, only plugins listed here can be added to this plugin.

See also: *parent_classes*

disable_child_plugins   Default: `False`

Disables dragging of child plugins in structure mode.

frontend_edit_template   Default: `cms/toolbar/placeholder_wrapper.html`

The template used for wrapping the plugin in frontend editing.

See also: *change_form_template*

model   Default: `CMSPlugin`

If the plugin requires per-instance settings, then this setting must be set to a model that inherits from `CMSPlugin`.

See also: *Storing configuration*

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.

See also: *child_classes*, *parent_classes*, *require_parent*,

parent_classes   Default: `None`

A list of Plugin Class Names. If this is set, this plugin may only be added to plugins listed here.

See also: *child_classes*, *require_parent*

render_plugin Default: `True`

Should the plugin be rendered at all, or doesn't it have any output? If *render_plugin* is `True`, then you must also define `render_template()`

See also: *render_template*, *get_render_template*

render_template Default: `None`

The path to the template used to render the template. If `render_plugin` is `True` either this or `get_render_template` **must** be defined;

See also: *render_plugin* , *get_render_template*

require_parent Default: `False`

Is it required that this plugin is a child of another plugin? Or can it be added to any placeholder, even one attached to a page.

See also: *child_classes*, *parent_classes*

text_enabled Default: `False`

Can the plugin be inserted inside the text plugin? If this is `True` then `icon_src()` must be overridden.

See also: *icon_src*, *icon_alt*

Methods

render 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.

If you overwrite this method it's recommended to always populate the context with default values by calling the render method of the super class:

```python
def render(self, context, instance, placeholder):
    context = super(MyPlugin, self).render(context, instance, placeholder)
    ...
    return context
```

get_render_template If you need to determine the plugin render model at render time you can implement `get_render_template()` method on the plugin class; this method taks the same arguments as `render`. The method **must** return a valid template file path.

Example:

```
def get_render_template(self, context, instance, placeholder):
    if instance.attr = 'one':
        return 'template1.html'
    else:
        return 'template2.html'
```

See also: *render_plugin* , *render_template*

icon_src By default, this returns an empty string, which, if left unoverridden would result in no icon rendered at all, which, in turn, would render the plugin uneditable by the operator inside a parent text plugin.

Therefore, this should be overridden when the plugin has `text_enabled` set to `True` to return the path to an icon to display in the text of the text plugin.

icon_src takes 1 argument:

  • `instance`: The instance of the plugin model

Example:

```
def icon_src(self, instance):
    return settings.STATIC_URL + "cms/img/icons/plugins/link.png"
```

See also: *text_enabled*, *icon_alt*

icon_alt Although it is optional, authors of "text enabled" plugins should consider overriding this function as well.

This function accepts the `instance` as a parameter and returns a string to be used as the alt text for the plugin's icon which will appear as a tooltip in most browsers. This is useful, because if the same plugin is used multiple times within the same text plugin, they will typically all render with the same icon rendering them visually identical to one another. This alt text and related tooltip will help the operator distinguish one from the others.

By default `icon_alt()` will return a string of the form: "[plugin type] - [instance]", but can be modified to return anything you like.

`icon_alt()` takes 1 argument:

  • `instance`: The instance of the plugin model

The default implementation is as follows:

```
def icon_alt(self, instance):
    return "%s - %s" % (force_text(self.name), force_text(instance))
```

See also: *text_enabled*, *icon_src*

text_editor_button_icon When *text_enabled* is `True`, this plugin can be added in a text editor and there might be an icon button for that purpose. This method allows to override this icon.

By default, it returns `None` and each text editor plugin may have its own fallback icon.

`text_editor_button_icon()` takes 2 arguments:

  • `editor_name`: The plugin name of the text editor

  • `icon_context`: A dictionary containing information about the needed icon like *width*, *height*, *theme*, etc

Usually this method should return the icon url. But, it may depends on the text editor because what is needed may differ. Please consult the documentation of your text editor plugin.

This requires support from the text plugin; support for this is currently planned for djangocms-text-ckeditor 2.5.0.

See also: *text_enabled*

---

get_extra_placeholder_menu_items `get_extra_placeholder_menu_items(self, request, placeholder)`

Extends the context menu for all placeholders. To add one or more custom context menu items that are displayed in the context menu for all placeholders when in structure mode, override this method in a related plugin to return a list of `cms.plugin_base.PluginMenuItem` instances.

get_extra_global_plugin_menu_items `get_extra_global_plugin_menu_items(self, request, plugin)`

Extends the context menu for all plugins. To add one or more custom context menu items that are displayed in the context menu for all plugins when in structure mode, override this method in a related plugin to return a list of `cms.plugin_base.PluginMenuItem` instances.

get_extra_local_plugin_menu_items `get_extra_local_plugin_menu_items(self, request, plugin)`

Extends the context menu for a specific plugin. To add one or more custom context menu items that are displayed in the context menu for a given plugin when in structure mode, override this method in the plugin to return a list of `cms.plugin_base.PluginMenuItem` instances.

CMSPlugin Attributes and Methods Reference

These are a list of attributes and methods that can (or should) be overridden on your plugin's *model* definition.

See also: *Storing configuration*

Attributes

translatable_content_excluded_fields Default: `[ ]`

A list of plugin fields which will not be exported while using `get_translatable_content()`.

See also: *get_translatable_content*, *set_translatable_content*

Methods

copy_relations Handle copying of any relations attached to this plugin. Custom plugins have to do this themselves.

`copy_relations` takes 1 argument:

- `old_instance`: The source plugin instance

See also: *Handling Relations*, *post_copy*

get_translatable_content Get a dictionary of all content fields (field name / field value pairs) from the plugin.

Example:

```python
from djangocms_text_ckeditor.models import Text

plugin = Text.objects.get(pk=1).get_plugin_instance()[0]
plugin.get_translatable_content()
# returns {'body': u'<p>I am text!</p>\n'}
```

See also: *translatable_content_excluded_fields*, *set_translatable_content*

---

post_copy   Can (should) be overridden to handle the copying of plugins which contain children plugins after the original parent has been copied.

`post_copy` takes 2 arguments:

- `old_instance`: The old plugin instance instance
- `new_old_ziplist`: A list of tuples containing new copies and the old existing child plugins.

See also: *Handling Relations*, *copy_relations*

set_translatable_content   Takes a dictionary of plugin fields (field name / field value pairs) and overwrites the plugin's fields. Returns `True` if all fields have been written successfully, and `False` otherwise.

set_translatable_content takes 1 argument:

- `fields`: A dictionary containing the field names and translated content for each.

Example:

```python
from djangocms_text_ckeditor.models import Text

plugin = Text.objects.get(pk=1).get_plugin_instance()[0]
plugin.set_translatable_content({'body': u'<p>This is a different text!</p>\n'})
# returns True
```

See also: *translatable_content_excluded_fields*, *get_translatable_content*

add_url   Returns the url to call to add a plugin instance; useful to implement plugin-specific logic in a custom view

Default: None (`cms_page_add_plugin` view is used)

edit_url   Returns the url to call to edit a plugin instance; useful to implement plugin-specific logic in a custom view

Default: None (`cms_page_edit_plugin` view is used)

move_url   Returns the url to call to move a plugin instance; useful to implement plugin-specific logic in a custom view

Default: None (`cms_page_move_plugin` view is used)

delete_url   Returns the url to call to delete a plugin instance; useful to implement plugin-specific logic in a custom view

Default: None (`cms_page_delete_plugin` view is used)

copy_url   Returns the url to call to copy a plugin instance; useful to implement plugin-specific logic in a custom view

Default: None (`cms_page_copy_plugins` view is used)

5.4.4 API References

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.

> **Warning:** Due to potential circular dependency issues, it's recommended to import the api in the functions
> that uses its function.
> e.g. use:

```python
def my_function():
    from cms.api import api_function

    api_function(...)
```

instead of:

```python
from cms.api import api_function

def my_function():
    api_function(...)
```

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 visi-
> bility to authenticated users.

cms.api.**VISIBILITY_ANONYMOUS**

> Used for the `limit_menu_visibility` keyword argument to *create_page()*. Limits menu visi-
> bility to anonymous (not authenticated) users.

cms.api.**create_page**(*title*, *template*, *language*, *menu_title=None*, *slug=None*, *apphook=None*,
apphook_namespace=None, *redirect=None*, *meta_description=None*,
created_by='python-api', *parent=None*, *publication_date=None*, *pub-
lication_end_date=None*, *in_navigation=False*, *soft_root=False*, *re-
verse_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
> > - **apphook_namespace** (*string*) – Name of the apphook namespace
> > - **redirect** (*string*) – URL redirect
> > - **meta_description** (*string*) – Description Of 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** (*bool*) – Whether this page should be in the navigation or not
- **soft_root** (*bool*) – 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** (*bool*) – Whether this page should be published or not
- **site** (django.contrib.sites.models.Site instance) – Site to put this page on
- **login_required** (*bool*) – Whether users must be logged in or not to view this page
- **limit_menu_visibility** (*VISIBILITY_ALL* or *VISIBILITY_USERS* or *VISIBILITY_ANONYMOUS*) – 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*, *redirect=None*, *meta_description=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*
- **redirect** (*string*) – URL redirect
- **meta_description** (*string*) – Description of 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 instance) – Placeholder to add the plugin to
- **plugin_type** (string or *cms.plugin_base.CMSPluginBase* subclass, must be a valid plugin) – What type of plugin to add
- **language** (*string*) – Language code for this plugin, must be in *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_*** (`bool`) – Permissions to give the user
- **grant_all** (`bool`) – 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** (`bool`) – Grant all permissions to the user

`cms.api.`**`publish_page`**(*page*, *user*, *language*)

Publishes a page.

> **Parameters**
> - **page** (`cms.models.pagemodel.Page` instance) – The page to publish
> - **user** (`django.contrib.auth.models.User` instance) – The user that performs this action
> - **language** (`string`) – The target language to publish to

`cms.api.`**`publish_pages`**(*include_unpublished=False*, *language=None*, *site=None*)

Publishes multiple pages defined by parameters.

> **Parameters**
> - **include_unpublished** (`bool`) – Set to `True` to publish all drafts, including unpublished ones; otherwise, only already published pages will be republished
> - **language** (`string`) – If given, only pages in this language will be published; otherwise, all languages will be published
> - **site** (`django.contrib.sites.models.Site` instance) – Specify a site to publish pages for specified site only; if not specified pages from all sites are published

`get_page_draft(page)`:

Returns the draft version of a page, regardless if the passed in page is a published version or a draft version.

> **Parameters page** (`cms.models.pagemodel.Page` instance) – The page to get the draft version
> **Return page** draft version of the page

`copy_plugins_to_language(page, source_language, target_language, only_empty=True)`:

Copy the plugins to another language in the same page for all the page placeholders.

By default plugins are copied only if placeholder has no plugin for the target language; use `only_empty=False` to change this.

> **Warning:** This function skips permissions checks

> **Parameters**
> - **page** (`cms.models.pagemodel.Page` instance) – the page to copy
> - **source_language** (`string`) – The source language code, must be in LANGUAGES

- **target_language** (*string*) – The source language code, must be in LANGUAGES
- **only_empty** (*bool*) – if False, plugin are copied even if plugins exists in the target language (on a placeholder basis).

Return int number of copied plugins

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:

```python
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')
```

cms.constants

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

cms.constants.**LEFT**
    Used as a position indicator in the toolbar.

cms.constants.**RIGHT**
    Used as a position indicator in the toolbar.

cms.constants.**REFRESH**
    Constant used by the toolbar.

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.

   get_plugin_urls(*instance*)
        Returns URL patterns for which the plugin wants to register views for. They are included under django CMS PageAdmin in the plugin path (e.g.: /admin/cms/page/plugin/<plugin-name>/ in the default case). Useful if your plugin needs to asynchronously talk to the admin.

   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*.
>
> It's recommended to always populate the context with default values by calling the render method of the super class:

```python
def render(self, context, instance, placeholder):
    context = super(MyPlugin, self).render(context, instance, placeholder)
    ...
    return context
```

> **Parameters**
> * **context** – Current template context.
> * **instance** – Plugin instance that is being rendered.
> * **placeholder** – Name of the placeholder the plugin is in.
> **Return type** dict

cms.toolbar

All methods taking a `side` argument expect either *cms.constants.LEFT* or *cms.constants.RIGHT* for that argument.

Methods accepting the `position` argument can insert items at a specific position. This can be either `None` to insert at the end, an integer index at which to insert the item, a *cms.toolbar.items.ItemSearchResult* to insert it *before* that search result or a *cms.toolbar.items.BaseItem* instance to insert it *before* that item.

cms.toolbar.toolbar

class cms.toolbar.toolbar.**CMSToolbar**
> The toolbar class providing a Python API to manipulate the toolbar. Note that some internal attributes are not documented here.
>
> All methods taking a `position` argument expect either *cms.constants.LEFT* or *cms.constants.RIGHT* for that argument.
>
> This class inherits *cms.toolbar.items.ToolbarMixin*, so please check that reference as well.
>
> **is_staff**
> > Whether the current user is a staff user or not.
>
> **edit_mode**
> > Whether the toolbar is in edit mode.
>
> **build_mode**
> > Whether the toolbar is in build mode.

---

show_toolbar
    Whether the toolbar should be shown or not.

csrf_token
    The CSRF token of this request

toolbar_language
    Language used by the toolbar.

watch_models
    A list of model this toolbar works on; used for redirections after editing (*Detecting url changes*).

add_item(*item*, *position=None*)
    Low level API to add items.

    Adds an item, which must be an instance of `cms.toolbar.items.BaseItem`, to the toolbar.

    This method should only be used for custom item classes, as all builtin item classes have higher level APIs.

    Read above for information on `position`.

remove_item(*item*)
    Removes an item from the toolbar or raises a `KeyError` if it's not found.

get_or_create_menu(*key. verbose_name*, *side=LEFT*, *position=NOne*)
    If a menu with `key` already exists, this method will return that menu. Otherwise it will create a menu for that `key` with the given `verbose_name` on `side` at `position` and return it.

add_button(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *extra_wrapper_classes=None*, *side=LEFT*, *position=None*)
    Adds a button to the toolbar. `extra_wrapper_classes` will be applied to the wrapping `div` while `extra_classes` are applied to the `<a>`.

add_button_list(*extra_classes=None*, *side=LEFT*, *position=None*)
    Adds an (empty) button list to the toolbar and returns it. See `cms.toolbar.items.ButtonList` for further information.

cms.toolbar.items

class `cms.toolbar.items.`**ItemSearchResult**
    Used for the find APIs in *ToolbarMixin*. Supports addition and subtraction of numbers. Can be cast to an integer.

item
    The item found.

index
    The index of the item.

class `cms.toolbar.items.`**ToolbarMixin**
    Provides APIs shared between `cms.toolbar.toolbar.CMSToolbar` and *Menu*.

    The `active` and `disabled` flags taken by all methods of this class specify the state of the item added.

    `extra_classes` should be either `None` or a list of class names as strings.

REFRESH_PAGE
    Constant to be used with `on_close` to refresh the current page when the frame is closed.

LEFT
    Constant to be used with `side`.

RIGHT
    Constant to be used with `side`.

get_item_count()
> Returns the number of items in the toolbar or menu.

add_item(*item*, *position=None*)
> Low level API to add items, adds the `item` to the toolbar or menu and makes it searchable. `item` must be an instance of *BaseItem*. Read above for information about the `position` argument.

remove_item(*item*)
> Removes `item` from the toolbar or menu. If the item can't be found, a `KeyError` is raised.

find_items(*item_type*, ***attributes*)
> Returns a list of *ItemSearchResult* objects matching all items of `item_type`, which must be a subclass of *BaseItem*, where all attributes in `attributes` match.

find_first(*item_type*, ***attributes*)
> Returns the first *ItemSearchResult* that matches the search or `None`. The search strategy is the same as in *find_items()*. Since positional insertion allows `None`, it's safe to use the return value of this method as the position argument to insertion APIs.

add_sideframe_item(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *on_close=None*, *side=LEFT*, *position=None*)
> Adds an item which opens `url` in the side frame and returns it.
>
> `on_close` can be set to `None` to do nothing when the side frame closes, *REFRESH_PAGE* to refresh the page when it closes or a URL to open once it closes.

add_modal_item(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *on_close=REFRESH_PAGE*, *side=LEFT*, *position=None*)
> The same as *add_sideframe_item()*, but opens the `url` in a modal dialog instead of the side frame.
>
> `on_close` can be set to `None` to do nothing when the side modal closes, *REFRESH_PAGE* to refresh the page when it closes or a URL to open once it closes.
>
> Note: The default value for `on_close` is different in *add_sideframe_item()* then in *add_modal_item()*

add_link_item(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *side=LEFT*, *position=None*)
> Adds an item that simply opens `url` and returns it.

add_ajax_item(*name*, *action*, *active=False*, *disabled=False*, *extra_classes=None*, *data=None*, *question=None*, *side=LEFT*, *position=None*)
> Adds an item which sends a POST request to `action` with `data`. `data` should be `None` or a dictionary, the CSRF token will automatically be added to it.
>
> If `question` is set to a string, it will be asked before the request is sent to confirm the user wants to complete this action.

class cms.toolbar.items.**BaseItem**(*position*)
> Base item class.

template
> Must be set by subclasses and point to a Django template

side
> Must be either *cms.constants.LEFT* or *cms.constants.RIGHT*.

render()
> Renders the item and returns it as a string. By default calls *get_context()* and renders *template* with the context returned.

get_context()
> Returns the context (as dictionary) for this item.

class cms.toolbar.items.**Menu**(*name*, *csrf_token*, *side=LEFT*, *position=None*)
> The menu item class. Inherits *ToolbarMixin* and provides the APIs documented on it.

The csrf_token must be set as this class provides high level APIs to add items to it.

get_or_create_menu(*key*, *verbose_name*, *side=LEFT*, *position=None*)
  The same as *cms.toolbar.toolbar.CMSToolbar.get_or_create_menu()* but adds the menu as a sub menu and returns a *SubMenu*.

add_break(*identifier=None*, *position=None*)
  Adds a visual break in the menu, useful for grouping items, and returns it. identifier may be used to make this item searchable.

class cms.toolbar.items.**SubMenu**(*name*, *csrf_token*, *side=LEFT*, *position=None*)
  Same as *Menu* but without the *Menu.get_or_create_menu()* method.

class cms.toolbar.items.**LinkItem**(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *side=LEFT*)
  Simple link item.

class cms.toolbar.items.**SideframeItem**(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *on_close=None*, *side=LEFT*)
  Item that opens url in side frame.

class cms.toolbar.items.**AjaxItem**(*name*, *action*, *csrf_token*, *data=None*, *active=False*, *disabled=False*, *extra_classes=None*, *question=None*, *side=LEFT*)
  An item which posts data to action.

class cms.toolbar.items.**ModalItem**(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *on_close=None*, *side=LEFT*)
  Item that opens url in the modal.

class cms.toolbar.items.**Break**(*identifier=None*)
  A visual break for menus. identifier may be provided to make this item searchable. Since breaks can only be within menus, they have no side attribute.

class cms.toolbar.items.**ButtonList**(*identifier=None*, *extra_classes=None*, *side=LEFT*)
  A list of one or more buttons.

  The identifier may be provided to make this item searchable.

add_item(*item*)
  Adds item to the list of buttons. item must be an instance of *Button*.

add_button(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*)
  Adds a *Button* to the list of buttons and returns it.

class cms.toolbar.items.**Button**(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*)
  A button to be used with *ButtonList*. Opens url when clicked.

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`.

5.4.5 Form and model fields

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.

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`.

class `cms.forms.fields.`**PageSmartLinkField**
> A field making use of `cms.forms.widgets.PageSmartLinkWidget`. This field will offer you a list of matching internal pages as you type. You can either pick one or enter an arbitrary url to create a non existing entry. Takes a *placeholder_text* argument to define the text displayed inside the input before you type. The widget uses an ajax request to try to find pages match. It will try to find case insensitive matches amongst public and published pages on the *title*, *path*, *page_title*, *menu_title* fields.

5.4.6 Template Tags

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.

static_placeholder

New in version 3.0.

The static_placeholder templatetag can be used anywhere in any template and is not bound to any page or model. It needs a name and it will create a placeholder that you can fill with plugins afterwards. The static_placeholder tag is normally used to display the same content on multiple locations or inside of apphooks or other 3rd party apps. Static_placeholder need to be published to show up on live pages.

Example:

```
{% load cms_tags %}

{% static_placeholder "footer" %}
```

> **Warning:** Static_placeholders are not included in the undo/redo and page history pages

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

Example:

```
{% static_placeholder "footer" or %}There is no content.{% endstatic_placeholder %}
```

By default, a static placeholder applies to *all* sites in a project.

If you want to make your static placeholder site-specific, so that different sites can have their own content in it, you can add the flag `site` to the templatetag to achieve this.

Example:

```
{% static_placeholder "footer" site or %}There is no content.{% endstatic_placeholder %}
```

Note that the Django "sites" framework *is* required and `SITE_ID` **must* be set* in `settings.py` for this (not to mention other aspects of django CMS) to work correctly.

render_placeholder

{% render_placeholder %} is used if you have a PlaceholderField in your own model and want to render it in the template.

The `render_placeholder` tag takes the following parameters:

- `PlaceholderField` instance
- `width` parameter for context sensitive plugins (optional)
- `language` keyword plus `language-code` string to render content in the specified language (optional)
- `language` keyword plus `language-code` string to render content in the specified language (optional)
- `as` keyword followed by `varname` (optional): the templatetag output can be saved as a context variable for later use.

The following example renders the my_placeholder field from the mymodel_instance and will render only the english plugins:

```
{% load cms_tags %}

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

New in version 3.0.2: 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:

```
{% render_placeholder mymodel_instance.my_placeholder as placeholder_content %}
<p>{{ placeholder_content }}</p>
```

When used in this manner, the placeholder will not be displayed for editing when the CMS is in edit mode.

render_uncached_placeholder

The same as `render_placeholder`, but the placeholder contents will not be cached or taken from the cache.

Arguments:

- `PlaceholderField` instance
- `width` parameter for context sensitive plugins (optional)
- `language` keyword plus `language-code` string to render content in the specified language (optional)
- `as` keyword followed by `varname` (optional): the templatetag output can be saved as a context variable for later use.

Example:

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

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 %}
```

show_uncached_placeholder

The same as *show_placeholder*, but the placeholder contents will not be cached or taken from the cache.

Arguments:

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

Example:

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

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 %}
```

page_url

Displays the URL of a page in the current language.

Arguments:

- page_lookup (see *page_lookup* for more information)
- language (optional)
- site (optional)
- as var_name (version 3.0 or later, optional; page_url can now be used to assign the resulting URL to a context variable var_name)

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.

New in version 3.0: page_url now supports the as argument. When used this way, the tag emits nothing, but sets a variable in the context with the specified name to the resulting value.

When using the as argument PageNotFound exceptions are always suppressed, regardless of the setting of DEBUG and the tag will simply emit an empty string in these cases.

Example:

```
{# Emit a 'canonical' tag when the page is displayed on an alternate url #}
{% page_url request.current_page as current_url %}{% if current_url and current_url != request.get
```

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"`, `"changed_date"`, `"changed_by"` (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_plugin_instances %}
    <div style="width: {{ plugin.width }}00px;">
        {% render_plugin plugin %}
    </div>
{% endfor %}
</div>
```

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

New in version 3.0.

render_plugin_block

This templatetag acts like the templatetag 'render_model_block' but with a plugin instead of a model as its target. This is used to link from a block of markup to a plugin's changeform in edit/preview mode.

This is useful for user interfaces that have some plugins hidden from display in edit/preview mode, but the CMS author needs to expose a way to edit them. It is also useful for just making duplicate or alternate means of triggering the change form for a plugin.

This would typically be used inside a parent-plugin's render template. In this example code below, there is a parent container plugin which renders a list of child plugins inside a NAV block, then the actual plugin contents inside a DIV.contentgroup-items block. In this example, the nav block is always shown, but the items are only shown once the corresponding navigation element is clicked. Adding this render_plugin_block makes it significantly more intuitive to edit a child plugins content, by double-clicking its nav item in edit mode.

Arguments:

- `plugin`

Example:

```
{% load cms_tags l10n %}

{% block section_content %}
<div class="contentgroup-container">
  <nav class="contentgroup">
    <div class="inner">
      <ul class="contentgroup-items">{% for child in children %}
        {% if child.enabled %}
          <li class="item{{ forloop.counter0|unlocalize }}">
            {% render_plugin_block child %}
            <a href="#item{{ child.id|unlocalize }}">{{ child.title|safe }}</a>
            {% endrender_plugin_block %}
          </li>{% endif %}
        {% endfor %}
      </ul>
    </div>
  </nav>

  <div class="contentgroup-items">{% for child in children %}
    <div class="contentgroup-item item{{ child.id|unlocalize }}{% if not forloop.counter0 %} activ
      {% render_plugin child %}
    </div>{% endfor %}
  </div>
</div>
{% endblock %}
```

New in version 3.0.

render_model

> **Warning:** `render_model` marks as safe the content of the rendered model attribute. This may be a security risk if used on fields which may contains non-trusted content. Be aware, and use the templatetag accordingly.

`render_model` is the way to add frontend editing to any Django model. It both render the content of the given attribute of the model instance and makes it clickable to edit the related model.

If the toolbar is not enabled, the value of the attribute is rendered in the template without further action.

If the toolbar is enabled, click to call frontend editing code is added.

By using this templatetag you can show and edit page titles as well as fields in standard django models, see *Frontend editing for Page and Django models* for examples and further documentation.

Example:

```
<h1>{% render_model my_model "title" "title,abstract" %}</h1>
```

This will render to:

```
<!-- The content of the H1 is the active area that triggers the frontend editor -->
<h1><div class="cms_plugin cms_plugin-myapp-mymodel-title-1">{{ my_model.title }}</div></h1>
```

Arguments:

- `instance`: instance of your model in the template

- `attribute`: the name of the attribute you want to show in the template; it can be a context variable name; it's possible to target field, property or callable for the specified model; when used on a page object this argument accepts the special `titles` value which will show the page **title** field, while allowing editing **title**, **menu title** and **page title** fields in the same form;

- `edit_fields` (optional): a comma separated list of fields editable in the popup editor; when templatetag is used on a page object this argument accepts the special `changelist` value which allows editing the pages **changelist** (items list);

- `language` (optional): the admin language tab to be linked. Useful only for [django-hvad](#) enabled models.

- `filters` (optional): a string containing chained filters to apply to the output content; works the same way as `filter` templatetag;

- `view_url` (optional): the name of a url that will be reversed using the instance `pk` and the `language` as arguments;

- `view_method` (optional): a method name that will return a URL to a view; the method must accept `request` as first parameter.

- `varname` (optional): the templatetag output can be saved as a context variable for later use.

> **Warning:** In this version of django CMS, the setting `CMS_UNESCAPED_RENDER_MODEL_TAGS` has a default value of `True` to provide behavior consistent with previous releases. However, all developers are encouraged to set this value to `False` to help prevent a range of security vulnerabilities stemming from HTML, Javascript, and CSS Code Injection.

> **Warning:** `render_model` is only partially compatible with django-hvad: using it with hvad-translated fields (say {% render_model object 'translated_field' %} return error if the hvad-enabled object does not exists in the current language. As a workaround `render_model_icon` can be used instead.

New in version 3.0.

render_model_block

`render_model_block` is the block-level equivalent of `render_model`:

```
{% render_model_block my_model %}
    <h1>{{ instance.title }}</h1>
    <div class="body">
        {{ instance.date|date:"d F Y" }}
        {{ instance.text }}
    </div>
{% endrender_model_block %}
```

This will render to:

```
<!-- This whole block is the active area that triggers the frontend editor -->
<div class="cms_plugin cms_plugin-myapp-mymodel-1">
    <h1>{{ my_model.title }}</h1>
    <div class="body">
        {{ my_model.date|date:"d F Y" }}
        {{ my_model.text }}
    </div>
</div>
```

In the block the `my_model` is aliased as `instance` and every attribute and method is available; also templatetags and filters are available in the block.

Arguments:

- `instance`: instance of your model in the template

- `edit_fields` (optional): a comma separated list of fields editable in the popup editor; when templatetag is used on a page object this argument accepts the special `changelist` value which allows editing the pages **changelist** (items list);

- `language` (optional): the admin language tab to be linked. Useful only for django-hvad enabled models.

- `view_url` (optional): the name of a url that will be reversed using the instance `pk` and the `language` as arguments;

- `view_method` (optional): a method name that will return a URL to a view; the method must accept `request` as first parameter.

- `varname` (optional): the templatetag output can be saved as a context variable for later use.

New in version 3.0.

render_model_icon

`render_model_icon` is intended for use where the relevant object attribute is not available for user interaction (for example, already has a link on it, think of a title in a list of items and the titles are linked to the object detail view); when in edit mode, it renders an **edit** icon, which will trigger the editing changeform for the provided fields.

```
<h3><a href="{{ my_model.get_absolute_url }}">{{ my_model.title }}</a> {% render_model_icon my_mo
```

It will render to something like:

```
<h3>
    <a href="{{ my_model.get_absolute_url }}">{{ my_model.title }}</a>
    <div class="cms_plugin cms_plugin-myapp-mymodel-1 cms_render_model_icon">
        <!-- The image below is the active area that triggers the frontend editor -->
        <img src="/static/cms/img/toolbar/render_model_placeholder.png">
    </div>
</h3>
```

Note: Icon and position can be customized via CSS by setting a background to the `.cms_render_model_icon img` selector.

Arguments:

- `instance`: instance of your model in the template

- `edit_fields` (optional): a comma separated list of fields editable in the popup editor; when templatetag is used on a page object this argument accepts the special `changelist` value which allows editing the pages **changelist** (items list);

- `language` (optional): the admin language tab to be linked. Useful only for django-hvad enabled models.

- `view_url` (optional): the name of a url that will be reversed using the instance `pk` and the `language` as arguments;

- `view_method` (optional): a method name that will return a URL to a view; the method must accept `request` as first parameter.

- `varname` (optional): the templatetag output can be saved as a context variable for later use.

New in version 3.0.

render_model_add

`render_model_add` is similar to `render_model_icon` but it will enable to create instances of the given instance class; when in edit mode, it renders an **add** icon, which will trigger the editing addform for the provided model.

```
<h3><a href="{{ my_model.get_absolute_url }}">{{ my_model.title }}</a> {% render_model_add my_mode
```

It will render to something like:

```
<h3>
    <a href="{{ my_model.get_absolute_url }}">{{ my_model.title }}</a>
    <div class="cms_plugin cms_plugin-myapp-mymodel-1 cms_render_model_add">
        <!-- The image below is the active area that triggers the frontend editor -->
        <img src="/static/cms/img/toolbar/render_model_placeholder.png">
    </div>
</h3>
```

Note: Icon and position can be customized via CSS by setting a background to the `.cms_render_model_add img` selector.

Arguments:

- `instance`: instance of your model, or model class to be added

- `edit_fields` (optional): a comma separated list of fields editable in the popup editor;

- `language` (optional): the admin language tab to be linked. Useful only for django-hvad enabled models.

- `view_url` (optional): the name of a url that will be reversed using the instance `pk` and the `language` as arguments;

- `view_method` (optional): a method name that will return a URL to a view; the method must accept `request` as first parameter.

- `varname` (optional): the templatetag output can be saved as a context variable for later use.

Warning: If passing a class, instead of an instance, and using `view_method`, please bear in mind that the method will be called over an **empty instance** of the class, so attributes are all empty, and the instance does not exists on the database.

New in version 3.1.

render_model_add_block

`render_model_add_block` is similar to `render_model_add` but instead of emitting an icon that is linked to the add model form in a modal dialog, it wraps arbitrary markup with the same "link". This allows the developer to create front-end editing experiences better suited to the project.

All arguments are identical to `render_model_add`, but the templatetag is used in two parts to wrap the markup that should be wrapped.

```
{% render_model_add_block my_model_instance %}<div>New Object</div>{% endrender_model_add_block %
```

It will render to something like:

```
<div class="cms_plugin cms_plugin-myapp-mymodel-1 cms_render_model_add">
  <div>New Object</div>
</div>
```

> **Warning:** You **must** pass an *instance* of your model as instance parameter. The instance passed could be an existing models instance, or one newly created in your view/plugin. It does not even have to be saved, it is introspected by the templatetag to determine the desired model class.

Arguments:

- `instance`: instance of your model in the template

- `edit_fields` (optional): a comma separated list of fields editable in the popup editor;

- `language` (optional): the admin language tab to be linked. Useful only for django-hvad enabled models.

- `view_url` (optional): the name of a url that will be reversed using the instance `pk` and the `language` as arguments;

- `view_method` (optional): a method name that will return a URL to a view; the method must accept `request` as first parameter.

- `varname` (optional): the templatetag output can be saved as a context variable for later use.

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 Internationalisation.

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 Internationalisation.

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" %}
...
```

Note: Be aware that you can not surround the cms_toolbar tag with block tags. The toolbar tag will render everything below it to collect all plugins and placeholders, before it renders itself. Block tags interfere with this.

5.4.7 Command Line Interface

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

```
python manage.py cms
```

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.

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!

cms copy-lang

The `copy-lang` subcommand can be used to copy content (titles and plugins) from one language to another. By default the subcommand copy content from the current site (e.g. the value of `SITE_ID`) and only if the target placeholder has no content for the specified language; using the defined options you can change this.

You must provide two arguments:

- `from_language`: the language to copy the content from;
- `to_language`: the language to copy the content to.

It accepts the following options

- `force-copy`: set to copy content even if a placeholder already has content; if set, copied content will be appended to the original one;
- `site`: specifiy a SITE_ID to operate on sites different from the current one;
- `verbose`: set for more verbose output.

Example:

```
cms copy-lang en de force-copy site=2 verbose
```

cms copy-site

The `copy-site` subcommand can be used to copy content (pages and plugins) from one site to another. The subcommand copy content from the `from_site` to `to_site`; please note that static placehoders are copied as they are shared across sites. The whole source tree is copied, in the root of the target website. Existing pages on the target website are not modified.

You must provide two arguments:

- `from_site`: the site to copy the content from;

- `to_site`: the site to copy the content to.

Example:

```
cms copy-site 1 2
```

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.

cms publisher_publish

If you want to publish many pages at once, this command can help you. By default, this command publishes drafts for all public pages.

It accepts the following options

- `unpublished`: set to publish all drafts, including unpublished ones; if not set, only already published pages will be republished.

- `language`: specify a language code to publish pages in only one language; if not specified, this command publishes all page languages;

- `site`: specify a site id to publish pages for specified site only; if not specified, this command publishes pages for all sites;

Example:

```
#publish drafts for public pages in all languages
publisher_publish

#publish all drafts in all pages
cms publisher_publish --unpublished

#publish drafts for public pages in deutsch
cms publisher_publish --language=de
```

```
#publish all drafts in deutsch
cms publisher_publish --unpublished --language=de

#publish all drafts in deutsch, but only for site with id=2
cms publisher_publish --unpublished --language=de --site=2
```

> **Warning:** This command publishes drafts. You should review drafts before using this command, because they will become public.

Maintenance and repair

fix-mptt

Occasionally, the MPTT tree can become corrupted (this is one of the reasons for our move away from MPTT to MP in django CMS 3.1). Typical symptoms include problems when trying to copy or delete plugins or pages.

Once a database has been migrated from MPTT to MP, there is no use for this command.

5.5 Development & community

django CMS is an open-source project, and relies on its community of users to keep getting better.

You can join us online:

- in our IRC channel, #django-cms, on irc.freenode.net
- on our django CMS users email list for **general** django CMS questions and discussion
- on our django CMS developers email list for discussions about the **development of django CMS**

You don't need to be an expert developer to make a valuable contribution - all you need is a little knowledge of the system, and a willingness to follow the contribution guidelines.

Remember that contributions to the documentation are highly prized, and key to the success of the django CMS project.

Development is led by a team of **core developers**, and under the overall guidance of a **technical board**.

All activity in the community is governed by our Code of Conduct.

5.5.1 Development of django CMS

django CMS is developed by a community of developers from across the world, with a wide range and levels of skills and expertise. Every contribution, however small, is valued.

As an open source project, anyone is welcome to contribute in whatever form they are able, which can include taking part in dicussions, filing bug reports, proposing improvements, contributing code or documentation, and testing the system - amongst others.

Divio AG

django CMS was released under a BSD licence in 2009. It was created at Divio AG of Zürich, Switzerland, by Patrick Lauber, who led its development for several years.

Divio remains thoroughly committed to django CMS both as a high-quality technical product and as a healthy open source project.

Divio's role in steering the project's development is formalised in the *django CMS technical board*, whose members are drawn both from key staff at Divio and other members of the django CMS community.

Divio hosts the django CMS project website and maintains overall control of the django CMS repository.

Core developers

Leading this process is a small team of core developers - people who have made and continue to make a significant contribution to the project, and have a good understanding not only of the code powering django CMS, but also the longer-term aims and directions of the project.

All core developers are volunteers.

Core developers have commit authority to django CMS's repository on GitHub. It's up to a core developer to say when a particular pull request should be committed to the repository.

Core developers also keep an eye on the `#django-cms` IRC channel on the Freenode network, and the django CMS users and django CMS developers email lists.

In addition to leading the development of the project, the core developers have an important role in fostering the community of developers who work with django CMS, and who create the numerous applications, plugins and other software that integrates with it.

Finally, the core developers are responsible for setting the tone of the community and helping ensure that it continues to be friendly and welcoming to all who wish to participate. The values and standards of the community are set out in its Code of Conduct.

Commit policy for core developers

Except in the case of very minor patches - for example, fixing typos in documentation - core developers are not expected to merge their own commits, but to follow good practice and have their work reviewed and merged by another member of the team.

Similarly, substantial patches with significant implications for the codebase from other members of the community should be reviewed and discussed by more than one core developer before being accepted.

Current core developers

- Angelo Dini http://github.com/finalangel
- Benjamin Wohlwend http://github.com/piquadrat
- Daniele Procida http://github.com/evildmp
- Iacopo Spalletti http://github.com/yakky
- Jonas Obrist http://github.com/ojii
- Martin Koistinen http://github.com/mkoistinen
- Paulo Alvarado http://github.com/czpython
- Patrick Lauber http://github.com/digi604
- Stefan Foulis http://github.com/stefanfoulis

Retired core developers

- Chris Glass http://github.com/chrisglass
- Øyvind Saltvik http://github.com/fivethreeo

Election of new core developers

New members of the core team are selected by the technical board.

Technical board

Historically, django CMS's development has been led by members of staff from Divio. It has been (and will continue to be) a requirement of the CMS that it meet Divio's needs.

However, as the software has matured and its user-base has dramatically expanded, it has become increasingly important also to reflect a wider range of perspectives in the development process. The technical board exists to help guarantee this.

Role

The role of the board is to maintain oversight of the work of the core team, to set key goals for the project and to make important decisions about the development of the software.

In the vast majority of cases, the team of core developers will be able to resolve questions and make decisions without the formal input of the technical board; where a disagreement with no clear consensus exists however, the board will make the necessary definitive decision.

The board is also responsible for making final decisions on the election of new core developers to the team, and - should it be necessary - the removal of developers who have retired, or for other reasons.

Composition of the board

The members of the technical board will include key developers from Divio and others in the django CMS development community - developers who work *with* django CMS, as well as developers *of* django CMS - in order to help ensure that all perspectives are represented in important decisions about the software and the project.

The board may also include representatives of the django CMS community who are not developers but who have a valuable expertise in key fields (user experience, design, content management, etc).

The current members of the technical board are:

- Angelo Dini
- Daniele Procida
- Iacopo Spalletti
- Jonas Obrist
- Martin Koistinen
- Matteo Larghi

The board will co-opt new members as appropriate.

5.5.2 Contributing to django CMS

Like every open-source project, django CMS is always looking for motivated individuals to contribute to its source code.

Key points:

> **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 **do not** 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.

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.

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.

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

Python 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.

HTML, CSS and JavaScript   As of django CMS 3.2, we will use spaces within frontend code, not tabs as previously. In the meantime, please continue using tabs - all tabs will be converted to spaces in a single commit for 3.2.

Frontend code should be formatted for readability. If in doubt, follow existing examples, or ask.

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 things together.

Running and writing tests is really important: a pull request that lowers our testing coverage will only be accepted with a very good reason; bug-fixing patches **must** demonstrate the bug with a test to avoid regressions and to check that the fix works.

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

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 simple, unfussy English. Elegance of style is a secondary consideration, and your prose can be improved later if necessary.

Documentation should be:

- written using valid Sphinx/restructuredText syntax (see below for specifics) and the file extension should be `.rst`

- written in English (we have standardised on British spellings)

- 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

- wrapped at 100 characters per line

Merging documentation is pretty fast and painless.

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.

Except for the tiniest of change, we recommend that you test them before submitting. Follow the same steps above to fork and clone the project locally. Next, create a virtualenv so you can install the documentation tools:

```
virtualenv djcms-docs-env
source djcms-docs-env/bin/activate
pip install sphinx sphinx_rtd_theme
```

Now you can `cd` into the `django-cms/docs` directory and build the documentation:

```
make html
open build/html/index.html
```

This allows you to review your changes in your local browser. After each change, be sure to rebuild the docs using `make html`. If everything looks good, then it's time to push your changes to Github and open a pull request.

Documentation structure

Our documentation is divided into the following main sections:

- Tutorials (`introduction`): step-by-step tutorials to get you up and running
- How-to guides (`how_to`): guides covering more advanced development
- Key topics (`topics`): explanations of key parts of the system
- Reference (`reference`): technical reference for APIs, key models and so on
- Development & community (`contributing`)
- Release notes & upgrade information (`upgrade`)
- (in progress Using django CMS (`user`): guides for *using* rather than setting up or developing for the CMS

Documentation markup

Sections   We use Python documentation conventions for section marking:

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

Inline markup

- **use backticks - ` `` ` - for:**
    - literals - ` ``cms.models.pagemodel`` `
    - filenames - `edit ``settings.py`` `
    - names of fields and other specific items in the Admin interface - `edit ``Redirect`` `
- **use emphasis - `*Home*` - around:**
    - the names of available options in or parts of the Admin - `the *Toolbar*`
    - the names of important modes or states - `switch to *Edit mode*`
    - values in or of fields - `enter *Home*`
- **use strong emphasis - `**` - around:**
    - buttons that perform an action - `hit **Save as draft**`

Rules for using technical words   There should be one consistent way of rendering any technical word, depending on its context. Please follow these rules:

- in general use, simply use the word as if it were any ordinary word, with no capitalisation or highlighting: "Your placeholder can now be used."
- at the start of sentences or titles, capitalise in the usual way: "Placeholder management guide"

- when introducing the term for the the first time, or for the first time in a document, you may highlight it to draw attention to it: "**Placeholders** are special model fields".

- when the word refers specifically to an object in the code, highlight it as a literal: "`Placeholder` methods can be overwritten as required" - when appropriate, link the term to further reference documentation as well as simply highlighting it.

References   Use absolute links to other documentation pages - `:doc:'/how_to/toolbar'` - rather than relative links - `:doc:'/../toolbar'`. This makes it easier to run search-and-replaces when items are moved in the structure.

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.

Frontend

We are using SASS/Compass for our styles. The files are located within `cms/static/cms/sass` and can be compiled using the compass command `compass watch cms/static/cms/` from within the django-cms root.

This will invoke the **config.rb** within `cms/static/cms/` using the predefined settings.

5.5.3 Code and project management

We use our GitHub project for managing both django CMS code and development activity.

This document describes how we manage tickets on GitHub. By "tickets", we mean GitHub issues and pull requests (in fact as far as GitHub is concerned, pull requests are simply a species of issue).

Issues

Raising an issue

> **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 **do not** 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.

Except in the case of security matters, of course, you're welcome to raise issues in any way that suits you - *on one of our email lists, or the IRC channel* or in person if you happen to meet another django CMS developer.

It's very helpful though if you don't just raise an issue by mentioning it to people, but actually file it too, and that means creating a new issue on GitHub.

There's an art to creating a good issue report.

The *Title* needs to be both succint and informative. "show_sub_menu displays incorrect nodes when used with soft_root" is helpful, whereas "Menus are broken" is not.

In the *Description* of your report, we'd like to see:

- how to reproduce the problem
- what you expected to happen
- what did happen (a traceback is often helpful, if you get one)

Getting your issue accepted

Other django CMS developers will see your issue, and will be able to comment. A core developer may add further comments, or a *label*.

The important thing at this stage is to have your issue *accepted*. This means that we've agreed it's a genuine issue, and represents something we can or are willing to do in the CMS.

You may be asked for more information before it's accepted, and there may be some discussion before it is. It could also be rejected as a *non-issue* (it's not actually a problem) or *won't fix* (addressing your issue is beyond the scope of the project, or is incompatible with our other aims).

Feel free to explain why you think a decision to reject your issue is incorrect - very few decisions are final, and we're always happy to correct our mistakes.

How we process tickets

Tickets should be:

- given a *status*
- marked with *needs*
- marked with a kind
- marked with the components they apply to
- marked with *miscellaneous other labels*
- commented

A ticket's *status* and *needs* are the most important of these. They tell us two key things:

- *status*: what stage the ticket is at
- *needs*: what next actions are required to move it forward

Needless to say, these labels need to be applied carefully, according to the rules of this system.

GitHub's interface means that we have no alternative but to use colours to help identify our tickets. We're sorry about this. We've tried to use colours that will cause the fewest issues for colour-blind people, so we don't use green (since we use red) or yellow (since we use blue) labels, but we are aware it's not ideal.

django CMS ticket processing system rules

- one and only one status **must** be applied to each ticket
- a healthy ticket (blue) **cannot** have any *critical needs* (red)
- when closed, tickets **must** have either a healthy (blue) or dead (black) status
- a ticket with *critical needs* **must not** have *non-critical needs* or *miscellaneous other* labels
- *has patch* and *on hold* labels imply a related pull request, which **must** be linked-to when these labels are applied
- *component*, *non-critical need* and *miscellaneous other* labels should be applied as seems appropriate

Status

The first thing we do is decide whether we accept the ticket, whether it's a pull request or an issue. An accepted status means the ticket is healthy, and will have a blue label.

Basically, it's good for open tickets to be healthy (blue), because that means they are going somewhere.

Important: Accepting a ticket means marking it as healthy, with one of the blue labels.

issues The bar for *status: accepted* is high. The status can be revoked at any time, and should be when appropriate. If the issue needs a *design decision*, *expert opinion* or *more info*, it can't be *accepted*.

pull requests When a pull request is accepted, it should become *work in progress* or (more rarely) *ready for review* or even *ready to be merged*, in those rare cases where a perfectly-formed and unimprovable pull request lands in our laps. As for issues, if it needs a *design decision*, *expert opinion* or *more info*, it can't be accepted.

No issue or pull request can have both a blue (accepted) and a red, grey or black label at the same time.

Preferably, the ticket should either be accepted (blue), rejected (black) or marked as having critical needs (red) *as soon as possible*. It's important that open tickets should have a clear status, not least for the sake of the person who submitted it so that they know it's being assessed.

Tickets should not be allowed to linger indefinitely with critical (red) needs. If the opinions or information required to accept the ticket are not forthcoming, the ticket should be declared unhealthy (grey) with *marked for rejection* and rejected (black) at the next release.

Needs

Critical needs (red) affect status.

Non-critical needs labels (pink) can be added as appropriate (and of course, removed as work progresses) to pull requests.

It's important that open tickets should have a clear needs labels, so that it's apparent what needs to be done to make progress with it.

Kinds and components

Of necessity, these are somewhat porous categories. For example, it's not always absolutely clear whether a pull request represents an enhancement or a bug-fix, and tickets can apply to multiple parts of the CMS - so do the best you can with them.

Other labels

backport, *blocker*, *has patch* or *easy pickings* labels should be applied as appropriate, to healthy (blue) tickets only/

Comments

At any time, people can comment on the ticket, of course. Although only core maintainers can change labels, anyone can suggest changing a label.

Label reference

Components and *kinds* should be self-explanatory, but *statuses*, *needs* and *miscellaneous other labels* are clarified below.

Statuses

A ticket's *status* is its position in the pipeline - its point in our workflow.

Every issue should have a status, and be given one as soon as possible. **An issue should have only one status applied to it**.

Many of these statuses apply equally well to both issues and pull requests, but some make sense only for one or the other

accepted   (issues only) The issue has been accepted as a genuine issue that needs to be addressed. Note that it doesn't necessarily mean we will do what the issue suggests, if it makes a suggestion - simply that we agree that there is an issue to be resolved.

non-issue   The issue or pull request are in some way mistaken - the 'problem' is in fact correct and expected behaviour, or the problems were caused by (for example) misconfiguration.

When this label is applied, an explanation must be provided in a comment.

won't fix   The issue or pull request imply changes to django CMS's design or behaviour that the core team consider incompatible with our chosen approach.

When this label is applied, an explanation must be provided in a comment.

marked for rejection   We've been unable to reproduce the issue, and it has lain dormant for a long time. Or, it's a pull request of low significance that requires more work, and looks like it might have been abandoned. These tickets will be closed when we make the next release.

When this label is applied, an explanation must be provided in a comment.

work in progress   (pull requests only) Work is on-going.

The author of the pull request should include "(work in progress)" in its title, and remove this when they feel it's ready for final review.

ready for review   (pull requests only) The pull request needs to be reviewed. (Anyone can review and make comments recommending that it be merged (or indeed, any further action) but only a core maintainer can change the label.)

ready to be merged   (pull requests only) The pull request has successfully passed review. Core maintainers should not mark their own code, except in the simplest of cases, as *ready to be merged*, nor should they mark any code as *ready to be merged* and then merge it themselves - there should be another person involved in the process.

When the pull request is merged, the label should be removed.

Needs

If an issue or pull request lacks something that needs to be provided for it to progress further, this should be marked with a "needs" label. A "needs" label indicates an *action* that should be taken in order to advance the item's status.

Critical needs   *Critical needs* (red) mean that a ticket is 'unhealthy' and won't be *accepted* (issues) or *work in progress*, *ready for review* or *ready to be merged* until those needs are addressed. In other words, no ticket can have both a blue and a red label.)

more info   Not enough information has been provided to allow us to proceed, for example to reproduce a bug or to explain the purpose of a pull request.

expert opinion The issue or pull request presents a technical problem that needs to be looked at by a member of the core maintenance team who has a special insight into that particular aspect of the system.

design decision The issue or pull request has deeper implications for the CMS, that need to be considered carefully before we can proceed further.

Non-critical needs A healthy (blue) ticket can have non-critical needs:

patch (issues only) The issue has been given a *status: accepted*, but now someone needs to write the patch to address it.

tests, docs (pull requests only) Code without docs or tests?! In django CMS? No way!

Other

has patch (issues only) A patch intended to address the issue exists. This doesn't imply that the patch will be accepted, or even that it contains a viable solution.

When this label is applied, a comment should cross-reference the pull request(s) containing the patch.

easy pickings An easy-to-fix issue, or an easy-to-review pull request - newcomers to django CMS development are encouraged to tackle *easy pickings* tickets.

blocker We can't make the next release without resolving this issue.

backport Any patch will should be backported to a previous release, either because it has security implications or it improves documentation.

on hold (pull requests only) The pull request has to wait for a higher-priority pull request to land first, to avoid complex merges or extra work later. Any *on hold* pull request is by definition *work in progress*.

When this label is applied, a comment should cross-reference the other pull request(s).

5.5.4 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.

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-1.6.txt

# run the test suite
# note that you must be in the django-cms directory when you do this,
# otherwise you'll get "Template not found" errors
cd django-cms
python develop.py test
```

It can take a few minutes to run. Note that the selenium tests included in the test suite require that you have Firefox installed.

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.

Problems running the tests

We are working to improve the performance and reliability of our test suite. We're aware of certain problems, but need feedback from people using a wide range of systems and configurations in order to benefit from their experience.

Please use the open issue #3684 Test suite is error-prone on our GitHub repository to report such problems.

If you can help *improve* the test suite, your input will be especially valuable.

OS X users In some versions of OS X, getttext needs to be installed so that it is available to Django. If you run the tests and find that various tests in `cms.tests.frontend` and `cms.tests.reversion_tests.ReversionTestCase` raise errors, it's likely that you have this problem.

A solution is:

```
brew install gettext && brew link --force gettext
```

(This requires the installation of Homebrew)

ERROR: test_copy_to_from_clipboard (cms.tests.frontend.PlaceholderBasicTests) You may find that a single frontend test raises an error. This sometimes happens, for some users, when the entire suite is run. To work around this you can invoke the test class on its own:

```
develop.py test cms.PlaceholderBasicTests
```

and it should then run without errors.

Advanced testing options

`develop.py` is the django CMS development helper script.

To use a different database, set the `DATABASE_URL` environment variable to a dj-database-url compatible value.

-h, --help
    Show help.

--version
    Show CMS version.

--user

> Specifies a custom user model to use for testing, the shell, or the server. The name must be in the format <app name>.<model name>, and the custom app must reside in the cms.test_utils.projects module.

develop.py test Runs the test suite. Optionally takes test labels as arguments to limit the tests which should be run. Test labels should be in the same format as used in `manage.py test`.

--xvfb

> Use a virtual X framebuffer for frontend testing, requires xvfbwrapper to be installed.
>
> With this option you won't need a physical display.

--parallel

> Runs tests in parallel, using one worker process per available CPU core.
>
> Cannot be used together with *develop.py test --failfast*.
>
> ---
>
> **Note:** The output of the worker processes will be shown interleaved, which means that you'll get the results from each worker process individually, which might cause confusing output at the end of the test run.
>
> ---

--failfast

> Stop running tests on the first failure or error.

develop.py timed test Run the test suite and print the ten slowest tests. Optionally takes test labels as arguments to limit the tests which should be run. Test labels should be in the same format as used in `manage.py test`.

develop.py isolated test Runs each test in the test suite in a new process, thus making sure that tests don't leak state. This takes a very long time to run. Optionally takes test labels as arguments to limit the tests which should be run. Test labels should be in the same format as used in `manage.py test`.

--parallel

> Same as *develop.py test --parallel*.

develop.py server Run a server locally for testing. This is similar to `manage.py runserver`.

--port `<port>`

> Port to bind to. Defaults to 8000.

--bind `<bind>`

> Interface to bind to. Defaults to 127.0.0.1.

--migrate

> Use migrations instead of plain syncdb.

application-name, migration-number

> Options to specify a single migration to migrate to. When using Django 1.6 it only works if –migrate option is specified.

develop.py shell Opens a Django shell. This is similar to `manage.py shell`.

develop.py compilemessages Compiles the po files to mo files. This is similar to `manage.py compilemessages`.

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.

5.5.5 Code of Conduct

Participation in the django CMS project is governed by a code of conduct.

The django CMS community is a pleasant one to be involved in for everyone, and we wish to keep it that way. Participants are expected to behave and communicate with others courteously and respectfully, whether online or in person, and to be welcoming, friendly and polite.

We will not tolerate abusive behaviour or language or any form of harassment.

Individuals whose behaviour is a cause for concern will be give a warning, and if necessary will be excluded from participation in official django CMS channels (email lists, IRC channels, etc) and events. The Django Software Foundation will also be informed of the issue.

Raising a concern

If you have a concern about the behaviour of any member of the django CMS community, please contact one of the members of the *core development team*.

Your concerns will be taken seriously, treated as confidential and investigated. You will be informed, in writing and as promptly as possible, of the outcome.

5.5.6 Branch policy

- **master**: this is the current stable release, the version released on PyPI.

- **support/3.0.x**: this will be our *next stable release*; this is the most appropriate branch for fixes and patches that will go into the next **master**

- **develop**: this will be *django CMS 3.1*; this is the most appropriate branch for more substantial features that will need team co-ordination

If in doubt, ask on the #django-cms IRC channel on freenode or the django-cms-developers email list!

5.6 Release notes & upgrade information

Some versions of django CMS present more complex upgrade paths than others, and some **require** you to take action. It is strongly recommended to read the release notes carefully when upgrading.

It goes without saying that you should **backup your database** before embarking on any process that makes changes to your database.

5.6.1 3.1.6 release notes

What's new in 3.1.6

Bug Fixes

- Fix cache settings

- Fix user lookup for view restrictions/page permissions when using raw id field

- Fixes an issue relating to uninstalling a namespaced application

- Adds "Can change page" permission

- Addresses security vulnerabilities in the *render_model* template tag that could lead to escalation of privileges or other security issues.

- Addresses a security vulnerability in the cms' usage of the messages framework

- Fixes security vulnerabilities in custom FormFields that could lead to escalation of privileges or other security issues.

Important: This version of django CMS introduces a new setting: CMS_UNESCAPED_RENDER_MODEL_TAGS with a default value of True. This default value allows upgrades to occur without forcing django CMS users to do anything, but, please be aware that this setting continues to allow known security vulnerabilities to be present. Due to this, the new setting is immediately deprecated and will be removed in a near-future release.

To immediately improve the security of your project and to prepare for future releases of django CMS and related addons, the project administrator should carefully review each use of the `render_model` template tags provided by django CMS. He or she is encouraged to ensure that all content which is rendered to a page using this template tag is cleansed of any potentially harmful HTML markup, CSS styles or Javascript. Once the administrator or developer is satisfied that the content is clean, he or she can add the "safe" filter parameter to the render_model template tag if the content should be rendered without escaping. If there is no need to render the content unescaped, no further action is required.

Once all template tags have been reviewed and adjusted where necessary, the administrator should set `CMS_UNESCAPED_RENDER_MODEL_TAGS = False` in the project settings. At that point, the project is more secure and will be ready for any future upgrades.

DjangoCMS Text CKEditor

Action required CMS 3.1.6 is not compatible with djangocms-text-ckeditor < 2.7.1. If you're using djangocms-text-ckeditor, please upgrade to 2.7.1 or up. Keep in mind that djangocms-text-ckeditor >= 2.8 is compatible only with CMS 3.2.x releases.

5.6.2 3.1.5 release notes

What's new in 3.1.5

Bug Fixes

- Fixed a tree corruption when pasting a nested plugin under another plugin.

- Improve CMSPluginBase.render documentation

- Fix CMSEditableObject context generation which generates to errors with django-classy-tags 0.7.1

- Fix error in toolbar when LocaleMiddleware is not used

- Move templates validation in app.ready

- Fix ExtensionToolbar when language is removed but titles still exists

- Fix pages menu missing on fresh install 3.1

- Fix incorrect language on placeholder text for redirect field

- Fix PageSelectWidget JS syntax

- Fix redirect when disabling toolbar

- Fix CMS_TOOLBAR_HIDE causes 'WSGIRequest' object has no attribute 'toolbar'

Treebeard corruption

Prior to 3.1.5 moving or pasting nested plugins would lead to some non-fatal tree corruptions, raising an error when adding plugins under the newly pasted plugins.

To fix these problems, upgrade to 3.1.5 and then run `manage.py cms fix-tree` command to repair the tree.

DjangoCMS Text CKEditor

Action required   CMS 3.1.5 is not compatible with djangocms-text-ckeditor < 2.7.1. If you're using djangocms-text-ckeditor, please upgrade to 2.7.1 or up. Keep in mind that djangocms-text-ckeditor >= 2.8 is compatible only with CMS 3.2.x releases.

5.6.3 3.1.4 release notes - Unreleased - Draft

What's new in 3.1.4

Bug Fixes

- Fix use structure migration

- Fix fix-tree command

- Remove some warning for Django 1.9

- Fix using scroll bar of plugin menu in Firefox & IE moves plugins

- Fix JS error when using PageSelectWidget

- Fix whitespace markup issues in draft mode

- Detect plugin migrations layout in tests

- Fix some treebeard corruption issues

Treebeard corruptions

Prior to 3.1.4 deleting pages may lead to some non-fatal tree corruptions when results in error 500 when publishing, deleting, moving pages. To fix these issues, upgrade to 3.1.4 and then executes `manage.py cms fix-tree` command to restore the correct data.

Thanks

TBD

5.6.4 3.1.3 release notes

What's new in 3.1.3

Bug Fixes

- Add missing migration
- Exclude PageUser manager from migrations
- Fix check for template instance in Django 1.8.x
- Fix error in PageField for Django 1.8
- Fix some Page tree bugs
- Declare Django 1.6.9 dependency in setup.py
- Make sure cache version returned is an int
- Fix issue preventing migrations to run on a new database (django 1.8)
- Fix get User model in 0010 migration
- Fix support for unpublished language pages
- Add documentation for plugins datamigration
- Fix getting request in _show_placeholder_for_page on Django 1.8
- Fix template inheritance order
- Fix xframe options inheritance order
- Fix placeholder inheritance order
- Fix language chooser template
- Relax html5lib versions
- Fix redirect when deleting a page
- Correct South migration error
- Correct validation on numeric fields in modal popups
- Exclude scssc from manifest
- Remove unpublished pages from menu
- Remove page from menu items for performance reason
- Fix reachability of pages with expired ancestors
- Don't try to modify an immutable QueryDict
- Only attempt to delete cache keys if there are some to be deleted
- Update documentation section
- Fix language chooser template
- Cast to int cache version
- Fix extensions copy when using duplicate page/create page type

Thanks

Many thanks community members who have submitted issue reports and especially to these GitHub users who have also submitted pull requests: basilelegal, gigaroby, ikudryavtsev, jokerejoker, josjevv, tomwardill.

5.6.5 3.1.2 release notes

What's new in 3.1.2

Bug Fixes

- Fix placeholder cache invalidation under some circumstances
- Update translations

5.6.6 3.1.1 release notes

What's new in 3.1.1

- Add Django 1.8 support
- Tutorial updates and improvements
- Add copy_site command
- Add setting to disable toolbar for anonymous users
- Add setting to hide toolbar when a URL is not handled by django CMS
- Add editorconfig configuration

Bug Fixes

- Fixed an issue where privileged users could be tricked into performing actions without their knowledge via a CSRF vulnerability.
- Fix issue with causes menu classes to be duplicated in advanced settings
- Fix issue with breadcrumbs not showing
- Fix issues with show_menu templatetags
- Fix an error in placeholder cache
- Fix get_language_from_request if POST and GET exists
- Minor documentation fixes
- Revert whitespace cleanup on flash player to fix it
- Correctly restore previous status of dragbars
- Fix an issue related to "Empty all" Placeholder feature
- Fix plugin sorting in py3
- Fix language-related issues when retrieving page URL
- Fix search results number and items alignment in page changelist
- Preserve information regarding the current view when applying the CMS decorator
- Fix errors with toolbar population
- Fix error with watch_models type

- Fix error with plugin breadcrumbs order
- Change the label "Save and close" to "Save as draft"
- Fix X-Frame-Options on top-level pages
- Fix order of which application urls are injected into urlpatterns
- Fix delete non existing page language
- Fix language fallback for nested plugins
- Fix render_model template tag doesn't show correct change list
- Fix Scanning for placeholders fails on include tags with a variable as an argument
- Fix handling of plugin position attribute
- Fix for some structureboard issues
- Pin South version to 1.0.2
- Pin Html5lib version to 0.999 until a current bug is fixed
- Make shift tab work correctly in submenu
- Fix language chooser template

Potentially backward incompatibile changes

The order in which the applications are injected is now based on the page depth, if you use nested apphooks, you might want to check that this does not change the behavior of your applications depending on applications urlconf greediness.

Thanks

Many thanks community members who have submitted issue reports and especially to these GitHub users who have also submitted pull requests: astagi, dirtycoder, doctormo, douwevandermeij, driesdesmet, furiousdave, ldgarcia, maqnouch, nikolas, northben, olarcheveque, pa0lin082, peterfarrell, sam-m888, sephii, stefanw, tim-graham, vstoykov.

A special thank you to vad and nostalgiaz for their support on Django 1.8 support

A special thank to Matt Wilkes and Sylvain Fankhauser for reporting the security issue.

5.6.7 3.1 release notes

django CMS 3.1 has been planned largely as a consolidation release, to build on the progress made in 3.0 and establish a safe, solid base for more ambitious work in the future.

In this release we have tried to maintain maximum backwards-compatibility, particularly for third-party applications, and endeavoured to identify and tidy loose ends in the system wherever possible.

> **Warning:** Upgrading from previous versions
> 3.1 introduces some changes that **require** action if you are upgrading from a previous version. Please read *Upgrading django CMS 3.0 to 3.1* for a step-by-step guide to the process of upgrading from 3.0 to 3.1.

What's new in 3.1

Switch from MPTT to MP

Since django CMS 2.0 we have relied on MPTT (Modified Preorder Tree Traversal) for efficiently handling tree structures in the database.

In 3.1, Django MPTT has been replaced by django-treebeard, to improve performance and reliability.

Over the years MPTT has proved not to be fast enough for big tree operations (>1000 pages); tree corruption, because of transactional errors, has also been a problem.

django-treebeard uses MP (Materialized Path). MP is more efficient and has more error resistance then MPTT. It should make working with and using django CMS better - faster and reliable.

Other than this, end users should not notice any changes.

Note: User feedback required

We require as much feedback as possible about the performance of django-treebeard in this release. Please let us know your experiences with it, especially if you encounter any problems.

Note: Backward incompatible change

While most of the low-level interface is very similar between `django-mptt` and `django-treebeard` they are not exactly the same. If any custom code needs to make use of the low-level interfaces of the page or plugins tree, please see the django-treebeard documentation for information on how to use equivalent calls in `django-treebeard`.

Note: Handling plugin data migrations

Please check *Plugin data migrations* for information on how to create migrations compatible with django CMS 3.0 and 3.1

Action required   Run *manage.py cms fix-mptt* before you upgrade.

Developers who use django CMS will need to run the schema and data migrations that are part of this release. Developers of third-party applications that relied on the Django MPTT that shipped with django CMS are advised to update their own applications so that they install it independently.

Dropped support for Django 1.4 and 1.5

Starting from version 3.1, django CMS runs on Django 1.6 and 1.7.

Action required   If you're still on an earlier version, you will need to install a newer one, and make sure that your third-party applications are also up-to-date with it before attempting to upgrade django CMS.

South is now an optional dependency

As Django South is now required for Django 1.6 only, it's marked as an optional dependency.

Action required   To install South along with django CMS use `pip install django-cms[south]`.

Migrations moved

Migrations directories have been renamed to conform to the new standard layout:

- Django 1.7 migrations: in the default `cms/migrations` and `menus/migrations` directories
- South migrations: in the `cms/south_migrations` and `menus/south_migrations` directories

Action required   South 1.0.2 or newer is required to handle the new layout correctly, so make sure you have that installed.

If you are upgrading from django CMS 3.0.x running on Django 1.7 you need to remove the old migration path from MIGRATION_MODULES settings.

Plugins migrations moving process

Core plugins are being changeed to follow the new convention for the migration modules, starting with **djangocms_text_ckeditor** 2.5 released together with django CMS 3.1.

Action required   Check the readme file of each plugin when upgrading to know the actions required.

Structure mode permission

A new Can use Structure mode* permission has been added.

Without this permission, a non-superuser will no longer have access to structure mode. This makes possible a more strict workflow, in which certain users are able to edit content but not structure.

This change includes a data migration that adds the new permission to any staff user or group with `cms.change_page` permission.

Action required   You may need to adjust these permissions once you have completed migrating your database.

Note that if you have existing users in your database, but are installing django CMS and running its migrations for the first time, you will need to grant them these permissions - they will not acquire them automatically.

Simplified loading of view restrictions in the menu

The system that loads page view restrictions into the menu has been improved, simplifying the queries that are generated, in order to make it faster.

Note: User feedback required

We require as much feedback as possible about the performance of this feature in this release. Please let us know your experiences with it, especially if you encounter any problems.

Toolbar API extension

The toolbar API has been extended to permit more powerful use of it in future development, including the use of "clipboard-like" items.

For an example of how this can be used, see the new Aldryn Blueprint application.

Per-namespace apphook configuration

django CMS provides a new API to define namespaced Apphook configurations.

Aldryn Apphooks Config has been created and released as a standard implementation to take advantage of this, but other implementations can be developed.

Improvements to the toolbar user interface

Some minor changes have been implemented to improve the toolbar user interface. The old **Draft/Live** switch has been replaced to achieve a more clear distinction between page states, and **Edit** and **Save as draft** buttons are now available in the toolbar to control the page editing workflow.

Placeholder language fallback default to True

`language_fallback` in *CMS_PLACEHOLDER_CONF* is `True` by default.

New templatetags

render_model_add_block   The family of *render_model* templatetags that allow Django developers to make any Django model editable in the frontend has been extended with `render_model_add_block`, which can offer arbitrary markup as the *Edit* icon (rather than just an image as previously).

render_plugin_block   Some user interfaces have some plugins hidden from display in edit/preview mode. `render_plugin_block` provides a way to expose them for editing, and also more generally provides an alternative means of triggering a plugin's change form.

Plugin table naming

Old-style plugin table names (for example, `cmsplugin_<plugin name>` are no longer supported. Relevant code has been removed.

Action required   Any plugin table name must be migrated to the standard (`<application name>_<table name>` layout.

`cms.context_processors.media` replaced by `cms.context_processors.cms_settings`

Action   required   Replace   the   `cms.context_processors.media`   with `cms.context_processors.cms_settings` in `settings.py`.

Upgrading django CMS 3.0 to 3.1

Preliminary steps

Before upgrading, please make sure that your current database is consistent and in a healthy state.

To ensure this, run two commands:

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

Make a copy of the database before proceeding further.

Settings update

- Change `cms.context_processors.media` to `cms.context_processors.cms_settings` in `TEMPLATE_CONTEXT_PROCESSORS`.

- Add `treebeard` to `INSTALLED_APPS`, and remove `mptt` if not required by other applications.

- If using Django 1.7 remove `cms` and `menus` from `MIGRATION_MODULES` to support the new migration layout.

- If migrating from Django 1.6 and below to Django 1.7, remove `south` from `installed_apps`.

- Eventually set `language_fallback` to `False` in *CMS_PLACEHOLDER_CONF* if you do not want language fallback behavior for placeholders.

Update the database

- Rename the custom plugin table names, either by applying relevant migrations (if any) or renaming the tables manually;

- The migration for MPTT to `django-treebeard` is handled by the django CMS migrations, thus apply migrations to update your database:

```
python manage.py migrate
```

5.6.8 3.0.16 release notes - Unreleased - Draft

What's new in 3.0.16

Bug Fixes

- Fix JS error when using PageSelectWidget

- Fix whitespace markup issues in draft mode

- Detect plugin migrations layout in tests

Thanks

TBD

5.6.9 3.0.15 release notes

What's new in 3.0.15

Bug Fixes

- Relax html5lib versions

- Fix redirect when deleting a page

- Correct South migration error

- Correct validation on numeric fields in modal popups

- Exclude scssc from manifest

- Remove unpublished pages from menu

- Remove page from menu items for performance reason
- Fix reachability of pages with expired ancestors
- Don't try to modify an immutable QueryDict
- Only attempt to delete cache keys if there are some to be deleted
- Update documentation section
- Fix language chooser template
- Cast to int cache version
- Fix extensions copy when using duplicate page/create page type

Thanks

Many thanks community members who have submitted issue reports and especially to these GitHub users who have also submitted pull requests: basilelegal.

5.6.10 3.0.14 release notes

What's new in 3.0.14

Bug Fixes

- Fixed an issue where privileged users could be tricked into performing actions without their knowledge via a CSRF vulnerability.
- Fix issue with causes menu classes to be duplicated in advanced settings
- Fix issue with breadcrumbs not showing
- Fix issues with show_menu templatetags
- Minor documentation fixes
- Fix an issue related to "Empty all" Placeholder feature
- Fix plugin sorting in py3
- Fix search results number and items alignment in page changelist
- Preserve information regarding the current view when applying the CMS decorator
- Fix X-Frame-Options on top-level pages
- Fix order of which application urls are injected into urlpatterns
- Fix delete non existing page language
- Fix language fallback for nested plugins
- Fix render_model template tag doesn't show correct change list
- Fix Scanning for placeholders fails on include tags with a variable as an argument
- Pin South version to 1.0.2
- Pin Html5lib version to 0.999 until a current bug is fixed
- Fix language chooser template

Potentially backward incompatibile changes

The order in which the applications are injected is now based on the page depth, if you use nested apphooks, you might want to check that this does not change the behavior of your applications depending on applications urlconf greediness.

Thanks

Many thanks community members who have submitted issue reports and especially to these GitHub users who have also submitted pull requests: douwevandermeij, furiousdave, nikolas, olarcheveque, sephii, vstoykov.

A special thank to Matt Wilkes and Sylvain Fankhauser for reporting the security issue.

5.6.11 3.0.13 release notes

What's new in 3.0.13

Bug Fixes

- Numerous documentation including installation and tutorial updates
- Numerous improvements to translations
- Improves reliability of apphooks
- Improves reliabiliy of Advanced Settings on page when using apphooks
- Allow page deletion after template removal
- Improves upstream caching accuracy
- Improves CMSAttachMenu registration
- Improves handling of mistyped URLs
- Improves redirection as a result of changes to page slugs, etc.
- Improves performance of "watched models"
- Improves frontend performance relating to resizing the sideframe
- Corrects an issue where items might not be visible in structue mode menus
- Limits version of django-mptt used in CMS for 3.0.x
- Prevent accidental upgrades to Django 1.8, which is not yet supported

Many thanks community members who have submitted issue reports and especially to these GitHub users who have also submitted pull requests: elpaso, jedie, jrief, jsma, treavis.

5.6.12 3.0.12 release notes

What's new in 3.0.12

Bug Fixes

- Fixes a regression caused by extra whitespace in Javascript

5.6.13 3.0.11 release notes

What's new in 3.0.11

- Core support for multiple instances of the same apphook'ed application
- The template tag *render_model_add* can now accept a model class as well as a model instance

Bug Fixes

- Fixes an issue with reverting to Live mode when moving plugins
- Fixes a missing migration issue
- Fixes an issue when using the PageField widget
- Fixes an issue where duplicate page slugs is not prevented in some cases
- Fixes an issue where copying a page didn't copy its extensions
- Fixes an issue where translations where broken when operating on a page
- Fixes an edge-case SQLite issue under Django 1.7
- Fixes an issue where a confirmation dialog shows only some of the plugins to be deleted when using the "Empty All" context-menu item
- Fixes an issue where deprecated 'mimetype' was used instead of 'contenttype'
- Fixes an issue where *cms check* erroneous displays warnings when a plugin uses class inheritance
- Documentation updates

Other

- Updated test CI coverage

5.6.14 3.0.10 release notes

What's new in 3.0.10

- Improved Py3 compatibility
- Improved the behavior when changing the operator's language
- Numerous documentation updates

Bug Fixes

- Revert a change that caused an issue with saving plugins in some browsers
- Fix an issue where urls were not refreshed when a page slug changes
- Fix an issue with FR translations
- Fixed an issue preventing the correct rendering of custom contextual menu items for plugins
- Fixed an issue relating to recovering deleted pages
- Fixed an issue that caused the uncached placeholder tag to display cached content
- Fixed an issue where extra slashed would appear in apphooked URLs when APPEND_SLASH=False
- Fixed issues relating to the logout function

5.6.15 3.0.9 release notes

What's new in 3.0.9

Bug Fixes

- Revert a change that caused a regression in toolbar login
- Fix an error in a translated phrase
- Fix error when moving items in the page tree

5.6.16 3.0.8 release notes

What's new in 3.0.8

- Add `require_parent` option to `CMS_PLACEHOLDER_CONF`

Bug Fixes

- Fix django-mptt version depenency to be PEP440 compatible
- Fix some Django 1.4 compatibility issues
- Add toolbar sanity check
- Fix behavior with CMSPluginBase.get_render_template()
- Fix issue on django >= 1.6 with page form fields.
- Resolve jQuery namespace issues in admin page tree and changeform
- Fix issues for PageField in Firefox/Safari
- Fix some Python 3.4 compatibility issue when using proxy modles
- Fix corner case in plugin copy
- Documentation fixes
- Minor code cleanups

> **Warning:** Fix for plugin copy patches a reference leak in `cms.models.pluginmodel.CMSPlugin.copy_plugins`, which caused the original plugin object to be modified in memory. The fixed code leaves the original unaltered and returns a modified copy. Custom plugins that called `cms.utils.plugins.copy_plugins_to` or `cms.models.pluginmodel.CMSPlugin.copy_plugins` may have relied on the incorrect behaviour. Check your code for calls to these methods. Correctly implemented calls should expect the original plugin instance to remain unaltered.

5.6.17 3.0.7 release notes

What's new in 3.0.7

- Numerous updates to the documentation
- Numerous updates to the tutorial
- Updates to better support South 1.0
- Adds some new, user-facing documentation

Bug Fixes

- Fixes an issue with placeholderadmin permissions

- Numerous fixes for minor issues with the frontend UI

- Fixes issue where the CMS would not reload pages properly if the URL contained a # symbol

- Fixes an issue relating to *limit_choices_to* in *forms.MultiValueFields*

- Fixes *PageField* to work in Django 1.7 environments

Project & Community Governance

- Updates to community and project governance documentation

- Added list of retired core developers

- Added branch policy documentaion

5.6.18 3.0.6 release notes

What's new in 3.0.6

Django 1.7 support

As of version 3.0.6 django CMS supports Django 1.7.

Currently our migrations for Django 1.7 are in `cms/migrations_django` to allow better backward compatibility; in future releases the Django migrations will be moved to the standard `migrations` directory, with the South migrations in `south_migrations`.

To support the current arrangement you need to add the following to your `settings`:

```
MIGRATION_MODULES = {
    'cms': 'cms.migrations_django',
    'menus': 'menus.migrations_django',
}
```

> **Warning: Applications migrations**
> **Any** application that defines a django CMS plugin or a model that uses a PlaceholderField or depends in any way on django CMS models **must** also provide Django 1.7 migrations.

Extended Custom User Support

If you are using custom user models and use `CMS_PERMISSION = True` then be sure to check that `PageUserAdmin` and `PageUserGroup` is still in working order.

The `PageUserAdmin` class now extends dynamically from the admin class that handles the user model. This allows us to use the same `search_fields` and filters in `PageUserAdmin` as in the custom user model admin.

CMSPlugin.get_render_template

A new method on plugins, that returns the template during the render phase, allowing you to change the template based on any plugin attribute or context status. See Custom Plugins for more.

Simplified toolbar API for page extensions

A simpler, more compact way to extend the toolbar for page extensions: *Simplified Toolbar API*.

5.6.19 3.0.3 release notes

What's new in 3.0.3

New Alias Plugin

A new Alias plugin has been added. You will find in your plugins and placeholders context menu in structure mode a new entry called "Create alias". This will create a new Alias plugin in the clipboard with a reference to the original. It will render this original plugin/placeholder instead. This is useful for content that is present in more then one place.

New Context Menu API

Plugins can now change the context menus of placeholders and plugins. For more details have a look at the docs:

Extending context menus of placeholders or plugins

Apphook Permissions

Apphooks have now by default the same permissions as the page they are attached to. This means if a page has for example a login required enabled all views in the apphook will have the same behavior.

Docs on how to disable or customize this behavior have a look here:

Apphook permissions

5.6.20 3.0 release notes

What's new in 3.0

> **Warning:** Upgrading from previous versions
> 3.0 introduces some changes that **require** action if you are upgrading from a previous version.

Note: *Click here to see the quick upgrade guide*

New Frontend Editing

django CMS 3.0 introduces a new frontend editing system as well as a customizable Django admin skin (djangocms_admin_style).

In the new system, `Placeholders` and their plugins are no longer managed in the admin site, but only from the frontend.

In addition, the system now offer two editing views:

- **content** view, for editing the configuration and content of plugins.
- **structure** view, in which plugins can be added and rearranged.

Page titles can also be modified directly from the frontend.

New Toolbar

The toolbar's code has been simplified and its appearance refreshed. The toolbar is now a more consistent management tool for adding and changing objects. See Extending the Toolbar.

> **Warning:** Upgrading from previous versions
> 3.0 now requires the `django.contrib.messages` application for the toolbar to work. See *Enable messages* for how to enable it.

New Page Types

You can now save pages as page types. If you then create a new page you may select a page type and all plugins and contents will be pre-filled.

Experimental Python 3.3 support

We've added experimental support for Python 3.3. Support for Python 2.5 has been dropped.

Better multilingual editing

Improvements in the django CMS environment for managing a multi-lingual site include:

- a built-in language chooser for languages that are not yet public.
- configurable behaviour of the admin site's language when switching between languages of edited content.

CMS_SEO_FIELDS

The setting has been **removed**, along with the SEO fieldset in admin.

- `meta_description` field's max_length is now 155 for optimal Google integration.
- `page_title` is default on top.
- `meta_keywords` field has been removed, as it no longer serves any purpose.

CMS_MENU_TITLE_OVERWRITE

New default for this setting is `True`.

Plugin fallback languages

It's now possible to specify fallback languages for a placeholder if the placeholder is empty for the current language. This must be activated in *CMS_PLACEHOLDER_CONF* per placeholder. It defaults to `False` to maintain pre-3.0 behavior.

language_chooser

The `language_chooser` template tag now only displays languages that are public. Use the toolbar language chooser to change the language to non-public languages.

Undo and Redo

If you have `django-reversion` installed you now have **undo** and **redo** options available directly in the toolbar. These can now revert *plugin* content as well as *page* content.

Plugins removed

We have removed plugins from the core. This is not because you are not expected to use them, but because django CMS should not impose unnecessary choices about what to install upon its adopters.

The most significant of these removals is `cms.plugins.text`.

We provide `djangocms-text-ckeditor`, a CKEditor-based Text Plugin. It's available from https://github.com/divio/djangocms-text-ckeditor. You may of course use your preferred editor; others are available.

Furthermore, we removed the following plugins from the core and moved them into separate repositories.

Note: In order to update from the old `cms.plugins.X` to the new `djangocms_X` plugins, simply install the new plugin, remove the old `cms.plugins.X` from `settings.INSTALLED_APPS` and add the new one to it. Then run the migrations (`python manage.py migrate djangocms_X`).

File Plugin We removed the file plugin (`cms.plugins.file`). Its new location is at:

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

As an alternative, you could also use the following (yet you will not be able to keep your existing files from the old `cms.plugins.file`!)

- https://github.com/stefanfoulis/django-filer

Flash Plugin We removed the flash plugin (`cms.plugins.flash`). Its new location is at:

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

Googlemap Plugin We removed the googlemap plugin (`cms.plugins.googlemap`). Its new location is at:

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

Inherit Plugin We removed the inherit plugin (`cms.plugins.inherit`). Its new location is at:

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

Picture Plugin We removed the picture plugin (`cms.plugins.picture`). Its new location is at:

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

Teaser Plugin We removed the teaser plugin (`cms.plugins.teaser`). Its new location is at:

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

Video Plugin We removed the video plugin (`cms.plugins.video`). Its new location is at:

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

Link Plugin We removed the link plugin (`cms.plugins.link`). Its new location is at:

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

Snippet Plugin We removed the snippet plugin (`cms.plugins.snippet`). Its new location is at:

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

As an alternative, you could also use the following (yet you will not be able to keep your existing files from the old `cms.plugins.snippet!`)

- https://github.com/pbs/django-cms-smartsnippets

Twitter Plugin Twitter disabled V1 of their API, thus we've removed the twitter plugin (`cms.plugins.twitter`) completely.

For alternatives have a look at these plugins:

- https://github.com/nephila/djangocms_twitter

- https://github.com/changer/cmsplugin-twitter

Plugin Context Processors take a new argument

Plugin Context have had an argument added so that the rest of the context is available to them. If you have existing plugin context processors you will need to change their function signature to add the extra argument.

Apphooks

Apphooks have moved from the title to the page model. This means you can no longer have separate apphooks for each language. A new `application instance name` field has been added.

Note: The reverse id is not used for the namespace anymore. If you used namespaced apphooks before, be sure to update your pages and fill out the namespace fields.

If you use apphook apps with `app_name` for app namespaces, be sure to fill out the instance namespace field `application instance name` as it's now required to have a namespace defined if you use app namespaces.

For further reading about application namespaces, please refer to the Django documentation on the subject at https://docs.djangoproject.com/en/dev/topics/http/urls/#url-namespaces

`request.current_app` has been removed. If you relied on this, use the following code instead in your views:

```
def my_view(request):
  current_app = resolve(request.path_info).namespace
  context = RequestContext(request, current_app=current_app)
  return render_to_response("my_template.html", context_instance=context)
```

Details can be found in *Attaching an application multiple times*.

PlaceholderAdmin

PlaceholderAdmin now is deprecated. Instead of deriving from `admin.ModelAdmin`, a new mixin class `PlaceholderAdminMixin` has been introduced which shall be used together with `admin.ModelAdmin`. Therefore when defining a model admin class containing a placeholder, now add `PlaceholderAdminMixin` to the list of parent classes, together with `admin.ModelAdmin`.

`PlaceholderAdmin` doesn't have language tabs anymore and the plugin editor is gone. The plugin API has changed and is now more consistent. `PageAdmin` uses the same API as `PlaceholderAdminMixin` now. If your app talked with the Plugin API directly be sure to read the code and the changed parameters. If you use `PlaceholderFields` you should add the mixin `PlaceholderAdminMixin` as it delivers the API for editing the plugins and the placeholders.

The workflow in the future should look like this:

1. Create new model instances via a toolbar entry or via the admin.

2. Go to the view that represents the model instance and add content via frontend editing.

Placeholder object permissions

In addition to model level permissions, `Placeholder` now checks if a user has permissions on a specific object of that model. Details can be found here in *Permissions*.

Placeholders are prefillable with default plugins

In CMS_PLACEHOLDER_CONF, for each placeholder configuration, you can specify via 'default_plugins' a list of plugins to automaticaly add to the placeholder if empty. See *default_plugins in CMS_PLACEHOLDER_CONF*.

Custom modules and plugin labels in the toolbar UI

It's now possible to configure module and plugins labels to show in the toolbar UI. See *CMS_PLACEHOLDER_CONF* for details.

New `copy-lang` subcommand

Added a management command to copy content (titles and plugins) from one language to another.

The command can be run with:

```
manage.py cms copy_lang from_lang to_lang
```

Please read *cms copy-lang* before using.

Frontedit editor for Django models

Frontend editor is available for any Django model; see *documentation* for details.

New `Page` `related_name` to `Site`

The `Page` object used to have the default `related_name` (page) to the `Site` model which may cause clashing with other Django apps; the `related_name` is now djangocms_pages.

> **Warning:** Potential backward incompatibility
> This change may cause you code to break, if you relied on `Site.page_set` to access cms pages from a `Site` model instance: update it to use `Site.djangocms_pages`

Moved all templatetags to cms_tags

All template tags are now in the `cms_tags` namespace so to use any cms template tags you can just do:

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

Getter and setter for translatable plugin content

A plugin's translatable content can now be read and set through `get_translatable_content()` and `set_translatable_content()`. See *Custom Plugins* for more info.

No more DB tablename magic for plugins

Since django CMS 2.0 plugins had their table names start with *cmsplugin_*. We removed this behavior in 3.0 and will display a deprecation warning with the old and new table name. If your plugin uses south for migrations create a new empty schemamigration and rename the table by hand.

> **Warning:** When working in the django shell or coding at low level, you **must** trigger the backward compatible behavior (a.k.a. magical rename checking), otherwise non migrated plugins will fail. To do this execute the following code:
>
> ```
> >>> from cms.plugin_pool import plugin_pool
> >>> plugin_pool.set_plugin_meta()
> ```
>
> This code can be executed both in the shell or in your python modules.

Added support for custom user models

Since Django 1.5 it has been possible to swap out the default User model for a custom user model. This is now fully supported by DjangoCMS, and in addition a new option has been added to the test runner to allow specifying the user model to use for tests (e.g. –user=customuserapp.User)

Page caching

Pages are now cached by default. You can disable this behavior with *CMS_PAGE_CACHE*

Placeholder caching

Plugins have a new default property: *cache=True*. If all plugins in a placeholder have set this to `True` the whole placeholder will be cached if the toolbar is not in edit mode.

> **Warning:** If your plugin is dynamic and processes current user or request data be sure to set `cache=False`

Plugin caching

Plugins have a new attribute: `cache=True`. Its default value can be configured with *CMS_PLUGIN_CACHE*.

Per-page Clickjacking protection

An advanced option has been added which controls, on a per-page basis, the `X-Frame-Options` header. The default setting is to inherit from the parent page. If no ancestor specifies a value, no header will be set, allowing Django's own middleware to handle it (if enabled).

CMS_TEMPLATE context variable

A new `CMS_TEMPLATE` variable is now available in the context: it contains the path to the current page template. See *CMS_TEMPLATE reference* for details.

Upgrading from 2.4

Note: There are reports that upgrading the CMS from 2.4 to 3.0 may fail if Django Debug Toolbar is installed. Please remove/disable Django Debug Toolbar and other non-essential apps before attempting to upgrade, then once complete, re-enable them.

If you want to upgrade from version 2.4 to 3.0, there's a few things you need to do. Start of by updating the cms' package:

```
pip install django-cms==3.0
```

Next, you need to make the following changes in your `settings.py`

- settings.INSTALLED_APPS
    - Remove `cms.plugin.twitter`. This package has been deprecated, see *Twitter Plugin*.
    - Rename all the other `cms.plugins.X` to `djangocms_X`, see *Plugins removed*.
- settings.CONTEXT_PROCESSORS
    - Replace `cms.context_processors.media` with `cms.context_processors.cms_settings`

Afterwards, install all your previously renamed ex-core plugins (djangocms-X). Here's a full list, but you probably don't need all of them:

```
pip install djangocms-file
pip install djangocms-flash
pip install djangocms-googlemap
pip install djangocms-inherit
pip install djangocms-picture
pip install djangocms-teaser
pip install djangocms-video
pip install djangocms-link
pip install djangocms-snippet
```

Also, please check your templates to make sure that you haven't put the `{% cms_toolbar %}` tag into a `{% block %}` tag. This is not allowed in 3.0 anymore.

To finish up, please update your database:

```
python manage.py syncdb
python manage.py migrate  (answer yes if your prompted to delete stale content types)
```

That's it!

Pending deprecations

placeholder_tags

`placeholder_tags` is now deprecated, the `render_placeholder` template tag can now be loaded from the `cms_tags` template tag library.

Using `placeholder_tags` will cause a `DeprecationWarning` to occur.

`placeholder_tags` will be removed in version 3.1.

cms.context_processors.media

`cms.context_processors.media` is now deprecated, please use `cms.context_processors.cms_settings` by updating `TEMPLATE_CONTEXT_PROCESSORS` in the settings

Using `cms.context_processors.media` will cause a `DeprecationWarning` to occur.

`cms.context_processors.media` will be removed in version 3.1.

5.6.21 2.4 release notes

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 MPTT 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 rather some unattractive monkey-patching of the `reverse()` function as well. As a benefit we now support localisation 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`:

```python
from django.conf.urls 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_REVISIONS* 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.

Backwards incompatible changes

New minimum requirements for dependencies

- Django 1.3 and Python 2.5 are no longer supported.

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.

5.6.22 2.3.4 release notes

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.

5.6.23 2.3.3 release notes

What's new in 2.3.3

Restored Python 2.5 support

2.3.3 restores Python 2.5 support for the django CMS.

Pending deprecations

Python 2.5 support will be dropped in django CMS 2.4.

5.6.24 2.3.2 release notes

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*.

5.6.25 2.3 release notes

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.

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)
```

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.

5.6.26 2.2 release notes

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.

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.

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 multi-linguality.

5.6.27 Upgrading from 2.1.x and Django 1.2.x

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
```

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 *Configuring your project for django CMS* 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
```

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>
```

Database Updates

Run the following commands to upgrade your database

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

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
```

5.7 Using django CMS

Note: This is a new section in the django CMS documentation, and a priority for the project. If you'd like to contribute to it, we'd love to hear from you - join us on the #django-cms IRC channel on freenode or the django-cms-developers email list.

The **Using django CMS** documentation is divided into two parts.

First, there's a *tutorial* that takes you step-by-step through key processes. Once you've completed this you will be familiar with the basics of content editing using the system.

The tutorial contains numerous links to items in the *reference section*.

The documentation in these two sections focuses on the basics of content creation and editing using django CMS's powerful front-end editing mode. It's suitable for non-technical and technical audiences alike.

However, it can only cover the basics that are common to most sites built using django CMS. Your own site will likely have many customisations and special purpose plugins which we cannot cover here. Nevertheless, by the end of this guide you should be comfortable with the content editing process using django CMS. Many of the skills you'll learn will be transferable to any custom plugins your site may have.

5.7.1 Tutorial

Note: This is a new section in the django CMS documentation, and a priority for the project. If you'd like to contribute to it, we'd love to hear from you - join us on the #django-cms IRC channel on freenode or the django-cms-developers email list.

It's strongly recommended that you follow this tutorial step-by-step. It has been designed to introduce you to the system in a methodical way, and each step builds on the previous one.

Log in

On a brand new site, you will see the default django CMS page:

The first step is to log into your site. You will need login credentials which are typically a username or email address plus a password. The developers of your site are responsible for creating and providing these credentials to you so consult them if you are unsure.

Your site will likely have a dedicated login page but a quick way to trigger the login form from any page is to simply append `?edit` to the url. Alternatively, hit *Switch to edit mode* on the default page).

This will reveal the *django CMS toolbar*, with a login prompt if you're not already logged-in:

And once you are logged in, the toolbar will display some key editing tools:

Create a page

Select the *Pages...* menu item from the *Site menu* (*example.com* in the example below, though yours may have a different name) in the toolbar.

This reveals the *side-frame* for page administration.

Hit **Add page**.

You're now asked for some *basic settings* for the new page.

Just give it a `Title` - call it "Home". You can ignore the rest for now.

You will notice that the `slug` is completed automatically, based on the `Title`.

When you are finished entering these fields, press **Save**. The *page editing form* is replaced with the

page list. It's a short list: you only have one item in it.

In the meantime to see the page you've just created, press *Home* in the page list; it'll be displayed in the main frame:

Create page content

Adding content to a page

Your page is empty of course. We need to add some content to it, by adding a *plugin*. In the toolbar, you'll notice that we're in *Content* mode. Change that to *Structure* mode, using the *Structure/Content button*.

This reveals the *placeholders* available on the page

On any placeholder, click the menu icon on the right side to reveal the list of available plugins. In this case, we'll choose the *Text* plugin. Invoking the *Text* plugin will open your installed text editor plugin. Enter some text and press **Save**.

When you save the plugin, your plugin will now be displayed "inside" the placeholder, as shown.

Previewing a page

To preview the page, switch back to *Content* mode using the *Structure/Content button* in the toolbar.

You can continue editing existing plugins in *Content* mode simply by double-clicking the content they present. Try it: double-click on the text you have just entered, and the text editor will open again for you to make some more changes.

To add new plugins, or to re-arrange existing ones, switch back into *Structure* mode.

Publish a page

The page you were working on in the previous step - Create page content - of this tutorial is not yet published. It's currently in *Draft* mode, and only you and other users with permission to edit the page can see it.

To publish it so that ordinary web visitors will be able to see it, hit **Publish page now**

Publish page now (a window alert will ask you for confirmation).

Using the toolbar you can make fur-

ther changes to the page: just select the **Edit** Edit button to activate *Edit mode*. Notice how the `?edit` and `?edit_off` parameters are appended to the URL in your browser as you switch in and out of *Edit* mode.

While your page remains in draft mode, you can continue working on it without affecting the

► CONTENT

► CONTENT   Quicksearch

File

Flash

Google Map

Inherit Plugins from Page

Link

Picture

Style

Teaser

Text

Video

5.7. Using django CMS

published
page.

Once
you have
finished
editing,
you can
exit edit-
ing mode
and save
the page
using
the **Save
as draft**
Save as draft
button.

You have now worked through the complete cycle of content publishing in django CMS.

Publishing status in the *Page list*

A page's status is also indicated in the *Page list* (if it's not already visible, select the *Pages...* menu item from the *Site menu*).

In the example below, the *Home* page is page is published (in English) but the *About* page is still only in *Draft* mode.

5.7.2 Reference for content editors

Note: This is a new section in the django CMS documentation, and a priority for the project. If you'd like to contribute to it, we'd love to hear from you - join us on the #django-cms IRC channel on freenode or the django-cms-developers email list.

Page admin

The interface

The django CMS toolbar   The toolbar is central to your content editing and management work in django CMS.

django CMS   Takes you back to home page of your site.

Site menu   *example.com* is the *Site menu* (and may have a different name for your site). Several options in this menu open up administration controls in the side-frame:

- *Pages* ... takes you directly to the pages editing interface
- *Users* ... takes you directly to the users management panel
- *Administration* ... takes you to the site-wide administration panel
- *User settings* ... allows you to switch the language of the admin interface and toolbar
- *Disable toolbar* allows you to completely disable the toolbar and front-end editing, regardless of login and staff status. To reactivate them, you need to enter *edit mode* either manually or through the backend administration.

You can also *Logout* from this menu.

Page menu   The *Page menu* contains options for managing the current page, and are either self-explanatory or will be described in a forthcoming documentation section.

History menu   Allows you to manage publishing and view publishing history of the current page.

Language **menu**   *Language* allows you to switch to a different language version of the page you're on, and manage the various translations.

Here you can:

- *Add* a missing translation
- *Delete* an existing translation
- *Copy* all plugins and their contents from an existing translation to the current one.

The *Structure/Content* button   Allows you to switch between different editing modes (when you're looking at a draft only).

Publishing controller   The *Publishing controller* manages the publishing state of your page - options are:

- **Publish page now** to publish an unpublished

- **Publish changes** to publish changes made to an existing page

- **Edit** to open the page for editing

- **Save as draft** to update the page and exit editing mode

The *disclosure triangle* A toggle to hide and reveal the toolbar.

The side-frame The *x* closes the side-frame. To reopen the side-frame, choose one of the links from the *Site menu* (named *example.com* by default).

The triangle icon expands and collapses the side-frame, and the next expands and collapses the main frame.

You can also adjust the side-frame's width by dragging it.

Admin views & forms

Page list The *page list* gives you an overview of your pages and their status. By default you get the basics:

The page you're currently on is highlighted in gray (in this case, *Journalism*, the last in the list).

From left to right, items in the list have:

- an *expand/collapse* control, if the item has children (*Home* and *Cheese* above)

- *tab* that can be used to drag and drop the item to a new place in the list

- the page's *Title*

- a *soft-root* indicator (*Cheese* has *soft-root* applied; *Home* is the menu root anyway)

- *language version* indicators and controls:

    - *blank*: the translation does not exist; pressing the indicator will open its *Basic settings* (in all other cases, hovering will reveal *Publish/Unpublish* options)

    - *grey*: the translation exists but is unpublished

    - *green*: the translation is published

    - *blue (pulsing)*: the translation has an amended draft

If you expand the width of the side-frame, you'll see more:

- *Menu* indicates whether the page will appear in navigation menus

- under *Actions*, options are:

    - *edit Basic settings*

    - *copy* page

    - *add child* (which can be placed before, after or below the page)

    - *cut* page

    - *delete* page

- *info* displays additional information about the page

Basic page settings To see a page's basic settings, select *Page settings...* from the *Page* menu. If your side-frame is wide enough, you can also use the *page edit icon* that appears in the *Actions* column in the page list view.

Required fields The page *Title* will typically be used by your site's

Change page

en fr de

Title:

Home

~~The default title~~

Slug:

templates, and displayed at the top of the page and in the browser's title bar and bookmarks. In this case search engines will use it too.

A *Slug* is part of the page's URL, and you'll usually want it to reflect the *Title*. In fact it will be generated automatically from the title, in an appropriate format - but it's always worth checking that your slugs are as short and sweet as possible.

Optional fields *Menu title* is used to override what is displayed in navigation menus - usually when the full *Title* is too long to be used there. For example, if the *Title* is "ACME Incorporated: Our story", it's going to be far too long to work well in the navigation menu, especially for your mobile users. "Our story" would be a more appropriate *Menu title*.

Page title is expected to be used by django CMS templates for the *<title>* element of the page (which will otherwise simply use the *Title* field). If provided, it will be the *Page title* that appears in the browser's title bar and bookmarks, and in search engine results.

Description meta tag is expected to be used to populate a *<meta>* tag in the document *<head>*. This is not displayed on the page, but is used for example by search engines for indexing and to show a summary of page content. It can also be used by other Django applications for similar purposes. Description is restricted to 155 characters, the number of characters search engines typcially use to show content.

Advanced settings A page's advanced settings are available by selecting *Advanced settings...* from the *Page* menu, or from the **Advanced settings** button at the bottom of the basic settings.

Most of the time it's not necessary to touch these settings.

- *Overwrite URL* allows you to change the URL from the default. By default, the URL for the page is the slug of the current page prefixed with slugs from parent pages. For example, the default URL for a page might be

/about/acme-incorporated/our-vision/. The *Overwrite URL* field allows you to shorten this to */our-vision/* while still keeping the page and its children organised under the *About* page in the navigation.

- *Redirect* allows you to redirect users to a different page. This is useful if you have moved content to another page but don't want to break URLs your users may have bookmarked or affect the rank of the page in search engine results.

- *Template* lets you set the template used by the current page. Your site will likely have a custom list of available templates. Templates are configured by developers to allow certain types of content to be entered into the page while still retaining a consistent layout.

- *Id* is an advanced field that should only be used in consultation with your site's developers. Changing this without consulting developers may result in a broken site.

- *Soft root* allows you to shorten the navigation hierarchy to something manageable on sites that have deeply nested pages. When selected, this page will act as the top-level page in the navigation.

- *Attached menu* allows you to add a custom menu to the page. This is typically used by developers to add custom menu logic to the current page. Changing this requires a server restart so should only be changed in consultation with developers.

- *Application* allows you to add custom applications (e.g. a weblog app) to the current page. This also is typically used by developers and requires a server restart to take effect.

- *X Frame Options* allows you to control whether the current page can be embedded in an iframe on another web page.

Working with admin in the frontend

The *Administration...* item in the *Site menu*, opens the *side-frame* containing the site's Django admin. This allows the usual interaction with the "traditional" Django admin.

Redirection

When an object is created or edited while the user is on the website frontend, a redirection occurs to redirect the user to the current address of the created/edited instance.

This redirection follows the rules below:

- an anonymous user (for example, after logging out) is always redirected to the home page

- when a model instance has changed (see *Detecting url changes*) the frontend is redirected to the instance URL, and:

  – in case of django CMS pages, the publishing state is taken into account, and then

Advanced Settings

en   fr   de

Overwrite URL: ☐

Keep this field empty if standard path should be used.

Redirect: ☐

Redirects to this URL.

Language independent options

Template: Inherit the template of the nearest an ⇕

The template used to render the content.

Id: ☐

A unique identifier that is used with the page_url templatetag for linking to this page

☐ Soft root

All ancestors will not be displayed in the navigation

Attached menu: --------- ⇕

Application: --------- ⇕

Hook application to this page.

X Frame Options: Inherit from parent page ⇕

Whether this page can be embedded in other pages or websites

* if the toolbar is in *Draft* mode the user is redirected to the *draft* page URL

* if in *Live* mode:

· the user is redirected to the page if is published

· otherwise it's switched in *Draft* mode and redirected to the *draft* page URL

• if the edited object or its URL can't be retrieved, no redirection occurs

Yes, it's complex - but there is a logic to it, and it's actually easier to understand when you're using it than by reading about it, so don't worry too much. The point is that django CMS always tries to redirect you to the most sensible place when it has to.

5.8 Indices and tables

• genindex

• modindex

• search

c

m

Symbols

A

B

C