django cms Documentation

Release 3.4.7

Divio AG and contributors

Jul 21, 2020

Contents

1 Overview **3**
- 1.1 Tutorials - start here . . . 3
- 1.2 How-to guides . . . 3
- 1.3 Key topics . . . 3
- 1.4 Reference . . . 3

2 Join us online **5**

3 Why django CMS? **7**

4 Software version requirements and release notes **9**
- 4.1 Django/Python compatibility table . . . 9

Python Module Index **259**

Index **261**


django
CMS

CHAPTER 1

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.

1.1 Tutorials - start here

For the new django CMS developer, from installation to creating your own addon applications.

1.2 How-to guides

Practical step-by-step guides for the more experienced developer, covering several important topics.

1.3 Key topics

Explanation and analysis of some key concepts in django CMS.

1.4 Reference

Technical reference material, for classes, methods, APIs, commands.

CHAPTER 2

Join us online

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

Our IRC channel, #django-cms, is on `irc.freenode.net`. If you don't have an IRC client, you can join our IRC channel using the KiwiIRC web client, which works pretty well.

Our django CMS users email list is for **general** django CMS questions and discussion

Our django CMS developers email list is for discussions about the **development of django CMS**

CHAPTER 3

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
- 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

CHAPTER 4

Software version requirements and release notes

This document refers to version 3.4.7.

4.1 Django/Python compatibility table

django CMS	Django						Python					
	1.4 & 1.5	1.6 & 1.7	1.8	1.9	1.10	1.11	2.6	2.7	3.3	3.4	3.5	3.6
3.0.18	✓	✓					✓	✓	✓	✓		
3.1.7		✓	✓				✓	✓	✓	✓		
3.2.0		✓	✓				✓	✓	✓	✓		
3.2.1-3.2.5		✓	✓	✓			✓	✓	✓	✓	✓	
3.3.0-3.4.1			✓	✓				✓	✓	✓	✓	
3.4.2-3.4.4			✓	✓	✓			✓	✓	✓	✓	
3.4.5			✓	✓	✓	✓		✓	✓	✓	✓	✓

See the repository's `setup.py` for more specific details of dependencies, or the *Release notes & upgrade information* for information about what is required or has changed in particular versions of the CMS.

The *installation how-to guide* provides an overview of other packages required in a django CMS project.

4.1.1 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 achieve 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 progression, 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.

Installing django CMS

We'll get started by setting up our environment.

Requirements

django CMS requires Django 1.8, 1.9 or 1.10 and Python 2.7, 3.3 or 3.4.

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
```

Update pip

`pip` is the Python installer. Make sure yours is up-to-date, as earlier versions can be less reliable:

```
pip install --upgrade pip
```

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 -f -p . mysite
```

This means:

- run the django CMS installer
- install Django Filer too (`-f`) - **required for this tutorial**
- use the current directory as the parent of the new project directory (`-p .`)
- call the new project directory `mysite`

Note: About Django Filer

Django Filer, a useful application for managing files and processing images. Although it's not required for django CMS itself, a vast number of django CMS addons use it, and nearly all django CMS projects have it installed. If you know you won't need it, omit the flag. See the django CMS installer documentation for more information.

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" %*
```

By default, the installer runs in Batch mode, and sets up your new project with some default values.

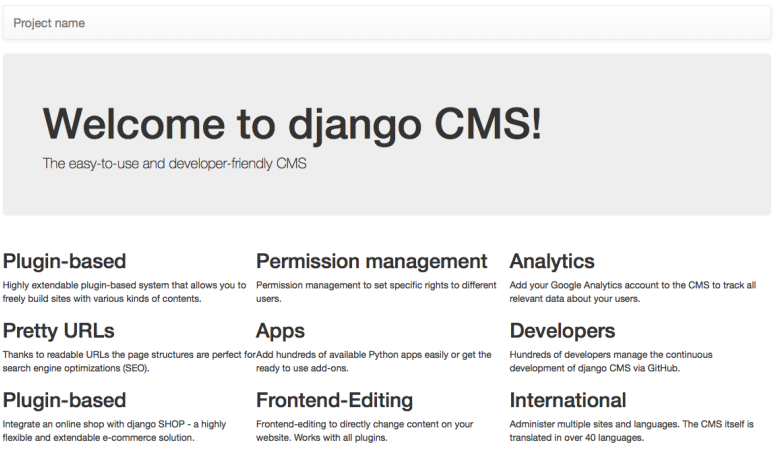
Later, you may wish to manage some of these yourself, in which case you need to run it in Wizard mode. The default in *Batch mode* is to set up an English-only project, which will be sufficient for the purposes of this tutorial. You can of course simply edit the new project's `settings.py` file at any time to change or add site languages or amend other settings.

The installer creates an admin user for you, with username/password `admin`/`admin`.

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.

If you are not already familiar with django CMS, take a few minutes to run through the basics of the *django CMS tutorial for users*.

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 `fullwidth.html`, the first one listed in the project's `settings.py` `CMS_TEMPLATES` tuple:

```
CMS_TEMPLATES = (
    ## Customize this
    ('fullwidth.html', 'Fullwidth'),
    ('sidebar_left.html', 'Sidebar Left'),
    ('sidebar_right.html', 'Sidebar Right')
)
```

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 template tags.

`fullwidth.html` contains a single placeholder, `{% placeholder "content" %}`.

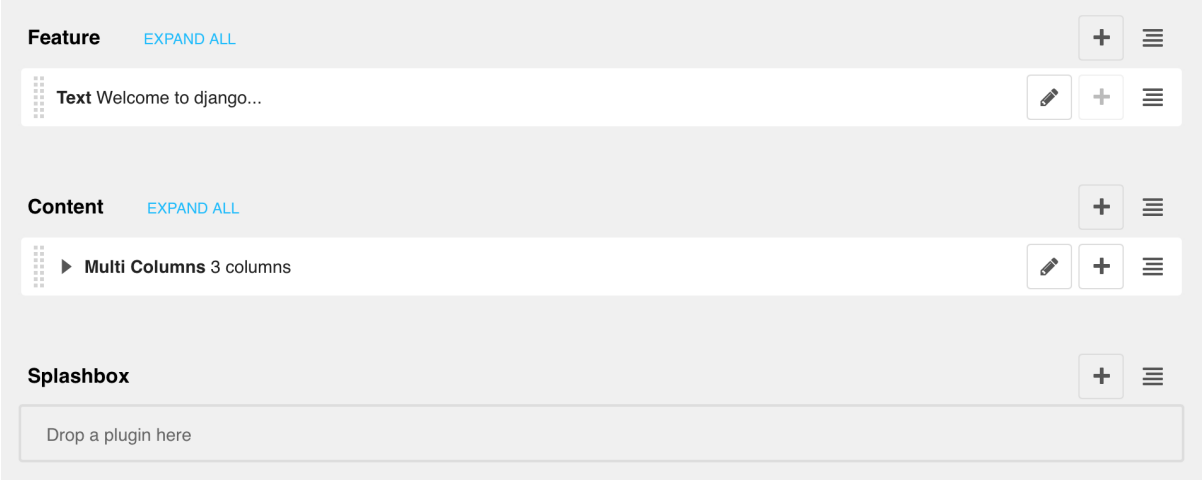
You'll also see `{% load cms_tags %}` in that file - `cms_tags` is the required template tag library.

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

Add a couple of new placeholders, `{% placeholder "feature" %}` and `{% placeholder "splashbox" %}` to the template's HTML structure. You can add them anywhere, for example:

```
{% block content %}
    {% placeholder "feature" %}
    {% placeholder "content" %}
    {% placeholder "splashbox" %}
{% endblock content %}
```

If you switch to *Structure* mode, you'll see the new placeholders available for use.



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* hard-code 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. Refresh any page in `Structure` mode, and you'll see the new static placeholder. If you add some content to it in the usual way, 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:

```
<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 *Integrating applications*.

Integrating applications

All the following sections of this tutorial are concerned with integrating other applications into django CMS, which is where a vast part of its power comes from.

Integrating applications doesn't just mean installing them alongside django CMS, so that they peacefully co-exist. It means using django CMS's features to build them into a single coherent web project that speeds up the work of managing the site, and makes possible richer and more automated publishing.

It's key to the way that django CMS integration works that **it doesn't require you to modify your other applications** unless you want to. This is particularly important when you're using third-party applications and don't want to have to maintain your own forked versions of them. (The only exception to this is if you decide to build django CMS features directly into the applications themselves, for example when using *placeholders in other applications*.)

For this tutorial, we're going to take a basic Django opinion poll application and integrate it into the CMS. So we'll install that, and create a second, independent, *Polls/CMS Integration* application to manage the integration, leaving the first untouched.

Install the `polls` application

Install the application from its GitHub repository using `pip`:

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

Let's add 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:

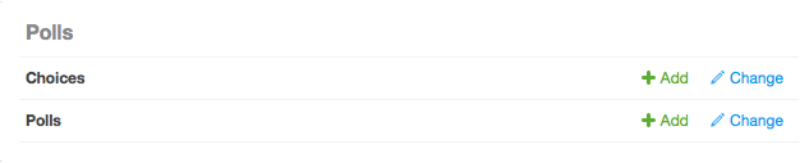
```
url(r'^', include('cms.urls')),
```

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 log in to the Django admin - `http://localhost:8000/admin/` - and find the Polls application.



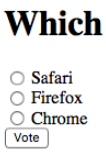
Create a new **Poll**, for example:

- **Question**: *Which browser do you prefer?*

  Choices:

  - *Safari*
  - *Firefox*
  - *Chrome*

Now if you visit `http://localhost:8000/en/polls/`, you should be able to see the published poll and submit a response.



Improve the templates for Polls

You'll have noticed that in the Polls application we only have minimal templates, and no navigation or styling.

Our django CMS pages on the other hand have access to a number of default templates in the project, all of which extend one called `base.html`. So, let's improve this by overriding the polls application's base template.

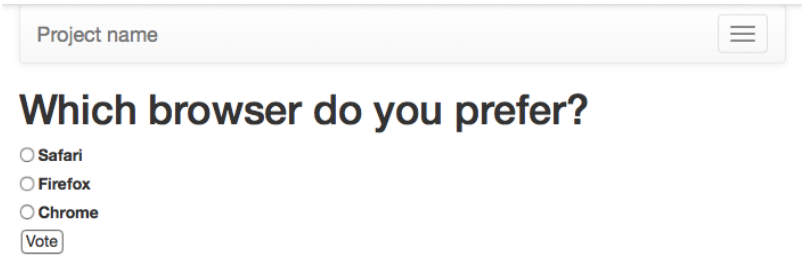
We'll do this in the *project* directory.

In `mysite/templates`, add `polls/base.html`, containing:

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

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

Refresh the `/polls/` page again, which should now be properly integrated into the site.



Set up a new `polls_cms_integration` application

So far, however, the Polls application has been integrated into the project, but not into django CMS itself. The two applications are completely independent. They cannot make use of each other's data or functionality.

Let's create the new *Polls/CMS Integration* application where we will bring them together.

Create the application

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

```
python manage.py startapp polls_cms_integration
```

So our workspace looks like this:

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

Add it to `INSTALLED_APPS`

Next is to integrate the `polls_cms_integration` application into the project.

Add `polls_cms_integration` to `INSTALLED_APPS` in `settings.py` - and now we're ready to use it to being integrating Polls with django CMS. We'll start by *developing a Polls plugin*.

Note: The project or the application?

Earlier, we added new templates to the project. We could equally well have have added `templates/polls/base.html` inside `polls_cms_integration`. After all, that's where we're going to be doing all the other integration work.

However, we'd now have an application that makes assumptions about the name of the template it should extend (see the first line of the `base.html` template we created) which might not be correct for a different project.

Also, we'd have to make sure that `polls_cms_integration` came *before* `polls` in `INSTALLED_APPS`, otherwise the templates in `polls_cms_integration` would not in fact override the ones in `polls`. Putting them in the project guarantees that they will override those in all applications.

Either way of doing it is reasonable, as long as you understand their implications.

Plugins

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

Install the `polls` application

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=polls
```

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

```
env/
    src/
        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:

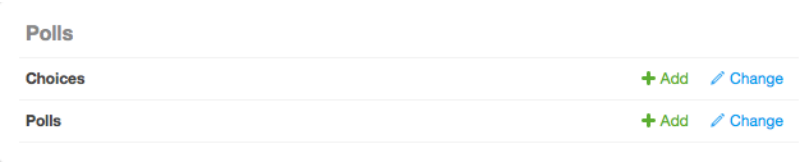
```
url(r'^', include('cms.urls')),
```

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 log in to the Django admin - `localhost:8000/admin/` - and find the Polls application.



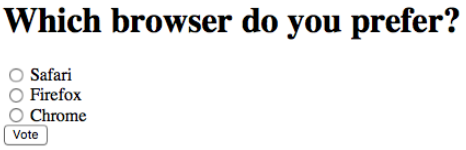
Create a new **Poll**, for example:

- **Question**: *Which browser do you prefer?*

  Choices:

  - *Safari*
  - *Firefox*
  - *Chrome*

Now if you visit `http://localhost:8000/en/polls/`, you should be able to see the published poll and submit a response.



Set up a base template for the application

However, in pages of the Polls application we only have minimal templates, and no navigation or styling.

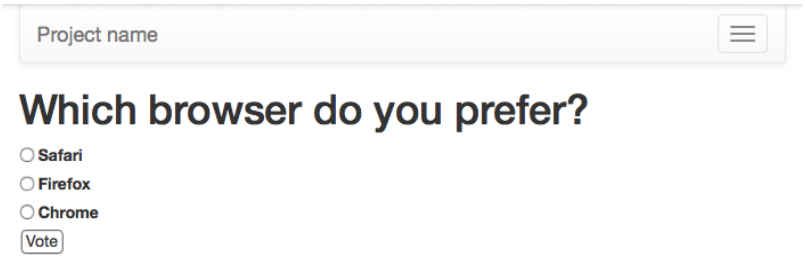
Let's improve this by overriding the polls application's base template.

In `mysite/templates`, add `polls/base.html`, containing:

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

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

Refresh the `/polls/` page again, which should now be properly integrated into the site.



So now we have integrated the standard polls application into our project.

Create a new `polls_cms_integration` application

So far, however, the polls application has been integrated into the project, but not into django CMS itself.

If you're already familiar with the CMS for a little, you'll have encountered django CMS *Plugins* - the objects you can place into placeholders on your pages: "Text", "Image" and so forth.

We're now going to extend the Django poll application so we can embed a poll easily into any CMS page. We'll put this integration code in a separate package, a *Polls/CMS Integration* application in our project.

Note: Why not build the plugin code into the polls application package?

This would certainly be possible, and in fact, if you were developing your own application it's what we would recommend. For a third-party application such as Polls however, placing the plugin code into a separate package means we don't have to modify or fork the original.

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

```
python manage.py startapp polls_cms_integration
```

So our workspace looks like this:

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

The Plugin Model

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

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


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

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

Note: django CMS plugins inherit from *`cms.models.pluginmodel.CMSPlugin`* (or a sub-class thereof) and not `models.Model`.

`PollPluginModel` might seem an odd choice for a model name (that is, with `model` in the name) but it helps distinguish it from the next class, `PollPluginPublisher`, that we need to create.

The Plugin Class

Now create a new 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:

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


@plugin_pool.register_plugin  # register the plugin
class PollPluginPublisher(CMSPluginBase):
    model = PollPluginModel  # model where plugin data are saved
    module = _("Polls")
    name = _("Poll Plugin")  # name of the plugin in the interface
    render_template = "polls_cms_integration/poll_plugin.html"

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

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

A reasonable convention for plugin naming is:

- `PollPluginModel`: the *model* class
- `PollPluginPublisher`: the *plugin* class

You don't need to follow this convention, but choose one that makes sense and stick to it.

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_cms_integration/templates/polls_cms_integration/poll_plugin.html` and should look something like this:

```
<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 %}
```

(continues on next page)

(continued from previous page)

```
    </div>
    <input type="submit" value="Vote" />
</form>
```

Integrate the `polls_cms_integration` application

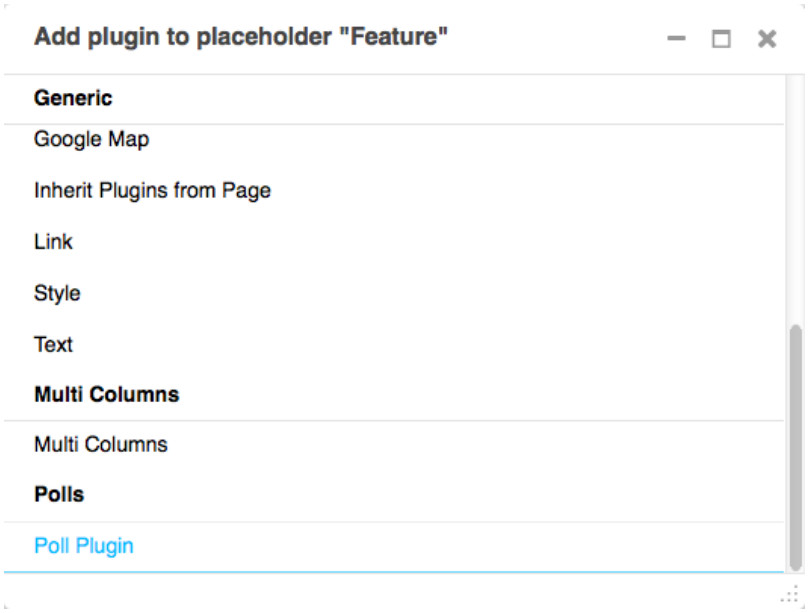
The final step is to integrate the `polls_cms_integration` application into the project.

Add `polls_cms_integration` to `INSTALLED_APPS` in `settings.py` and create a database migration to add the plugin table:

```
python manage.py makemigrations polls_cms_integration
python manage.py migrate polls_cms_integration
```

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.

Apphooks

Right now, our Django Polls application 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.

Create an apphook

We do this with an **apphook**, created using a *`CMSApp`* sub-class, which tells the CMS how to include that application.

Apphooks live in a file called `cms_apps.py`, so create one in your Polls/CMS Integration application, i.e. in `polls_cms_integration`.

This is a very basic example of an apphook for a django CMS application:

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


@apphook_pool.register  # register the application
class PollsApphook(CMSApp):
    app_name = "polls"
    name = _("Polls Application")

    def get_urls(self, page=None, language=None, **kwargs):
        return ["polls.urls"]
```

Instead of defining the URL patterns in another file `polls/urls.py`, it also is possible to return them directly, for instance as:

```
from django.conf.urls import url
from polls.views import PollListView, PollDetailView

class PollsApphook(CMSApp):
    # ...
    def get_urls(self, page=None, language=None, **kwargs):
        return [
            url(r'^$', PollListView.as_view()),
            url(r'^(?P<slug>[\w-]+)/?$', PollDetailView.as_view()),
        ]
```

What this all means

In the `PollsApphook` class, we have done several key things:

- The `app_name` attribute gives the system a way to refer to the apphook - see *Attaching an application multiple times* for details on why this matters.
- `name` is a human-readable name for the admin user.
- The `get_urls()` method is what actually hooks the application in, returning a list of URL configurations that will be made active wherever the apphook is used.

Restart the runserver. This is necessary because we have created a new file containing Python code that won't be loaded until the server restarts. You only have to do this the first time the new file has been created.

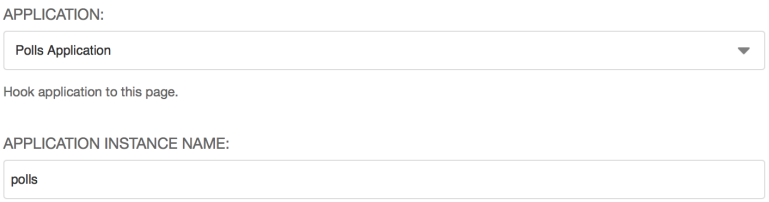
Apply the apphook to a page

Now we need to create a new page, and attach the Polls application to it through this apphook.

Create and save a new page, then publish it.

Note: Your apphook won't work until the page has been published.

In its *Advanced settings*, choose "Polls Application" from the *Application* menu, and save once more.



Refresh the page, and you'll find that the Polls application is now available directly from the new django CMS page.

You can now remove the mention of the Polls application (`url(r'^polls/', include('polls.urls', namespace='polls'))`) from your project's `urls.py` - it's no longer even required there.

Later, we'll install a django-CMS-compatible *third-party application*.

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.

Create the toolbar

We'll create a toolbar using a `cms.toolbar_base.CMSToolbar` sub-class.

Create a new `cms_toolbars.py` file in your Polls/CMS Integration application. Here's a basic example:

```
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


class PollToolbar(CMSToolbar):
    supported_apps = (
        'polls',
        'polls_cms_integration',
    )

    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'),
        )
```

(continues on next page)

(continued from previous page)

```
toolbar_pool.register(PollToolbar)  # register the toolbar
```

Note: Don't forget to restart the runserver to have your new toolbar item recognised.

What this all means

- `supported_apps` is a list of application names in which the toolbar should be active. Usually you don't need to set `supported_apps` - the appropriate application will be detected automatically. In this case (since the views for the Polls application are in `polls`, while our `cms_toolbars.py` is in the `polls_cms_integration` application) we need to specify both explicitly.
- `watch_models` allows the frontend editor to redirect the user to the model instance `get_absolute_url` whenever an instance of this model is created or saved through the frontend editor (see *Detecting URL changes* for details).
- The `populate()` method, which populates the toolbar menu with nodes, will only be called if the current user is a staff user. In this case it:
  - checks whether we're in a page belonging to this application, using `self.is_current_app`
  - ... if so, it creates a menu, if one's not already there (`self.toolbar.get_or_create_menu()`)
  - adds a menu item to list all polls in the overlay (`add_sideframe_item()`)
  - adds a menu item to add a new poll as a modal window (`add_modal_item()`)

See it at work

Visit your Polls page on your site, and you'll see a new *Polls* item in the toolbar.

It gives you quick access to the list of Polls in the Admin, and gives you a shortcut for creating a new Poll.

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

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.

Create the navigation menu

We create the menu using a *`CMSAttachMenu`* sub-class, and use the `get_nodes()` method to add the nodes.

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

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

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.

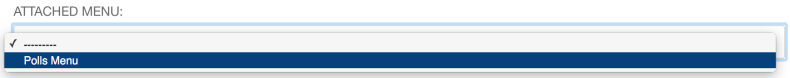
    def get_nodes(self, request):
        """
        This method is used to build the menu tree.
        """
        nodes = []
        for poll in Poll.objects.all():
            node = NavigationNode(
                title=poll.question,
                url=reverse('polls:detail', args=(poll.pk,)),
                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 won't actually do anything until attached to a page. In the *Advanced settings* of the page to which you attached the apphook earlier, select "Polls Menu" from the list of *Attached menu* options, and save once more. (You could add the menu to any page, but it makes most sense to add it to this page.)



You can force the menu to be added automatically to the page by the apphook if you consider this appropriate. See *Apphook menus* for information on how to do that.

Note: The point here is to illustrate the basic principles. In this actual case, note that:

- If you're going to use sub-pages, you'll need to improve the menu styling to make it work a bit better.
- Since the Polls page lists all the Polls in it anyway, this isn't really the most practical addition to the menu.

Content creation wizards

Content creation wizards allow you to make use of the toolbar's **Create** button in your own applications. It opens up a simple dialog box with the basic fields required to create a new item.

django CMS uses it for creating Pages, but you can add your own models to it.

In the `polls_cms_integration` application, add a `forms.py` file:

```
from django import forms

from polls.models import Poll


class PollWizardForm(forms.ModelForm):
    class Meta:
        model = Poll
        exclude = []
```

Then add a `cms_wizards.py` file, containing:

```
from cms.wizards.wizard_base import Wizard
from cms.wizards.wizard_pool import wizard_pool

from polls_cms_integration.forms import PollWizardForm


class PollWizard(Wizard):
    pass

poll_wizard = PollWizard(
    title="Poll",
    weight=200,  # determines the ordering of wizards in the Create dialog
    form=PollWizardForm,
    description="Create a new Poll",
)

wizard_pool.register(poll_wizard)
```

Refresh the Polls page, hit the **Create** button in the toolbar - and the wizard dialog will open, offering you a new wizard for creating Polls.

Note: Once again, this particular example is for illustration only. In the case of a Poll, with its multiple Questions associated with it via foreign keys, we really want to be able to edit those questions at the same time too.

That would require a much more sophisticated form and processing than is possible within the scope of this tutorial.

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 application, Aldryn News & Blog.

Basic installation

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

```
pip install aldryn-newsblog
```

Django settings

INSTALLED_APPS

Add the application 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 because you need to avoid duplication:

```
# you will probably need to add:
'aldryn_apphooks_config',
'aldryn_boilerplates',
'aldryn_categories',
'aldryn_common',
'aldryn_newsblog',
'aldryn_people',
'aldryn_reversion',
'djangocms_text_ckeditor',
'parler',
'sortedm2m',
'taggit',

# and you will probably find the following already listed:
'easy_thumbnails',
'filer',
'reversion',
```

THUMBNAIL_PROCESSORS

One of the dependencies is Django Filer. It provides a special feature that allows more sophisticated image cropping. For this to work it needs its own thumbnail processor (`filer.thumbnail_processors.scale_and_crop_with_subject_location`) to be listed in `settings.py` in place of `easy_thumbnails.processors.scale_and_crop`:

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

ALDRYN_BOILERPLATE_NAME

Aldryn News & Blog uses aldryn-boilerplates to provide multiple sets of templates and static files for different CSS frameworks. We're using the Bootstrap 3 in this tutorial, so let's choose `bootstrap3` by adding the setting:

```
ALDRYN_BOILERPLATE_NAME='bootstrap3'
```

STATICFILES_FINDERS

Add the boilerplates static files finder to `STATICFILES_FINDERS`, *immediately before* `django.contrib.staticfiles.finders.AppDirectoriesFinder`:

```
STATICFILES_FINDERS = [
    'django.contrib.staticfiles.finders.FileSystemFinder',
    '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 code above.

TEMPLATES

Important: In Django 1.8, the `TEMPLATE_LOADERS` and `TEMPLATE_CONTEXT_PROCESSORS` settings are rolled into the `TEMPLATES` setting. We're assuming you're using Django 1.8 here.

```
TEMPLATES = [
    {
        # ...
        'OPTIONS': {
            'context_processors': [
                # ...
                'aldryn_boilerplates.context_processors.boilerplate',
                ],
            'loaders': [
                # ...
                'aldryn_boilerplates.template_loaders.AppDirectoriesLoader',
                ],
            },
        },
    ]
```

Migrate the database

We've added a new application so we need to update our database:

```
python manage.py migrate
```

Start the server again.

Create a new apphooked page

The News & Blog application comes with a django CMS apphook, so add a new django CMS page (call it *News*), and add the News & Blog application to it *just as you did for Polls*.

For this application we also need to create and select an *Application configuration*.

Give this application configuration some settings:

- `Instance namespace`: *news* (this is used for reversing URLs)

- `Application title`: *News* (the name that will represent the application configuration in the admin)
- `Permalink type`: choose a format you prefer for news article URLs

Save this application configuration, and make sure it's selected in `Application configurations`.

Publish the new page, and you should find the News & Blog application at work there. (Until you actually create any articles, it will simply inform you that there are *No items available*.)

Add new News & Blog articles

You can add new articles using the admin or the new *News* menu that now appears in the toolbar when you are on a page belonging to News & Blog.

You can also insert a *Latest articles* plugin into another page - like all good django CMS applications, Aldryn News & Blog comes with plugins.

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 *How to install 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.

If you don't have an IRC client, you can join our IRC channel using the KiwiIRC web client, which works pretty well.

4.1.2 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.

How to install django CMS by hand

The easiest way to install django CMS is by using the automated django CMS installer. This is the recommended way to start with new projects, and it's what we use in the *tutorial section of this documentation*.

If you prefer to do things manually, this how-to guide will take you through the process.

Note: You can also use this guide to help you install django CMS as part of an existing project. However, the guide assumes that you are starting with a blank project, so you will need to adapt the steps below appropriately as required.

This document assumes you have some basic familiarity with Python and Django. After you've integrated django CMS into your project, you should be able to follow the *Tutorials* for an introduction to developing with django CMS.

Install the django CMS package

Check the *Python/Django requirements* for this version of django CMS.

django CMS also has other requirements, which it lists as dependencies in its `setup.py`.

Important: We strongly recommend doing all of the following steps in a virtual environment. You ought to know how to create, activate and dispose of virtual environments using virtualenv. If you don't, you can use the steps below to get started, but you are advised to take a few minutes to learn the basics of using virtualenv before proceeding further.

```
virtualenv django-cms-site  # create a virtualenv
source django-cms-site/bin/activate  # activate it
```

In an activated virtualenv, run:

```
pip install --upgrade pip
```

to make sure `pip` is up-to-date, as earlier versions can be less reliable.

Then:

```
pip install django-cms
```

to install the latest stable version of django CMS.

Create a new project

Create a new 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.

Your new project will look like this:

```
myproject
    myproject
        __init__.py
        settings.py
        urls.py
        wsgi.py
    manage.py
```

Minimally-required applications and settings

Open the new project's `settings.py` file in your text editor.

INSTALLED_APPS

You will need to add the following to its list of `INSTALLED_APPS`:

```
'django.contrib.sites',
'cms',
'menus',
'treebeard',
```

- django CMS needs to use Django's `django.contrib.sites` framework. You'll need to set a `SITE_ID` in the settings - `SITE_ID = 1` will suffice.
- `cms` and `menus` are the core django CMS modules.
- django-treebeard is used to manage django CMS's page and plugin tree structures.

django CMS installs django CMS admin style. This provides some styling that helps make django CMS administration components easier to work with. Technically it's an optional component and does not need to be enabled in your project, but it's strongly recommended.

In the `INSTALLED_APPS`, **before** `django.contrib.admin`, add:

```
'djangocms_admin_style',
```

Language settings

django CMS requires you to set the `LANGUAGES` setting. This should list all the languages you want your project to serve, and must include the language in `LANGUAGE_CODE`.

For example:

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

(For simplicity's sake, at this stage it is worth changing the default `en-us` in that you'll find in the `LANGUAGE_CODE` setting to `en`.)

Database

django CMS requires a relational database backend. Each django CMS installation should have its own database.

You can use SQLite, which is included in Python and doesn't need to be installed separately or configured further. You are unlikely to be using that for a project in production, but it's ideal for development and exploration, especially as it is configured by default in a new Django project's `DATABASES`.

Note: For deployment, you'll need to use a production-ready database with Django. We recommend using PostgreSQL or MySQL.

Installing and maintaining database systems is far beyond the scope of this documentation, but is very well documented on the systems' respective websites.

Whichever database you use, it will also require the appropriate Python adaptor to be installed:

```
pip install psycopg2      # for Postgres
pip install mysqlclient   # for MySQL
```

Refer to `Django's DATABASES setting documentation` for the appropriate configuration for your chosen database backend.

Database tables

Now run migrations to create database tables for the new applications:

```
python manage.py migrate
```

Admin user

Create an admin superuser:

```
python manage.py createsuperuser
```

Using `cms check` for configuration

Once you have completed the minimum required set-up described above, you can use django CMS's built-in `cms check` command to help you identify and install other components. Run:

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

This will check your configuration, your applications and your database, and report on any problems.

Note: If key components are be missing, django CMS will be unable to run the `cms check command` and will simply raise an error instead.

After each of the steps below run `cms check` to verify that you have resolved that item in its checklist.

Sekizai

Django Sekizai is required by the CMS for static files management. You need to have:

```
'sekizai'
```

listed in `INSTALLED_APPS`, and:

```
'sekizai.context_processors.sekizai'
```

in the `TEMPLATES['OPTIONS']['context_processors']`:

```
TEMPLATES = [
    {
        ...
        'OPTIONS': {
            'context_processors': [
                ...
                'sekizai.context_processors.sekizai',
            ],
        },
    },
]
```

Middleware

in your `MIDDLEWARE_CLASSES` you'll need `django.middleware.locale.LocaleMiddleware` - it's **not** installed in Django projects by default.

Also add:

```
'cms.middleware.user.CurrentUserMiddleware',
'cms.middleware.page.CurrentPageMiddleware',
'cms.middleware.toolbar.ToolbarMiddleware',
'cms.middleware.language.LanguageCookieMiddleware',
```

to the list.

You can also add `'cms.middleware.utils.ApphookReloadMiddleware'`. It's not absolutely necessary, but it's *useful*. If included, should be at the start of the list.

Context processors

Add `'cms.context_processors.cms_settings'` to `TEMPLATES['OPTIONS']['context_processors']`.

`cms check` should now be unable to identify any further issues with your project. Some additional configuration is required however.

Further required configuration

URLs

In the project's `urls.py`, add `url(r'^', include('cms.urls'))` to the `urlpatterns` list. It should come after other patterns, so that specific URLs for other applications can be detected first.

You'll also need to have an import for `django.conf.urls.include`. For example:

```
from django.conf.urls import url, include

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

The django CMS project will now run, as you'll see if you launch it with `python manage.py runserver`. You'll be able to reach it at http://localhost:8000/, and the admin at http://localhost:8000/admin/. You won't yet actually be able to do anything very useful with it though.

Templates

django CMS requires at least one template for its pages. The first template in the *`CMS_TEMPLATES`* list will be the project's default template.

```
CMS_TEMPLATES = [
    ('home.html', 'Home page template'),
]
```

In the root of the project, create a `templates` directory, and in that, `home.html`, a minimal django CMS template:

```
{% load cms_tags sekizai_tags %}
<html>
    <head>
        <title>{% page_attribute "page_title" %}</title>
        {% render_block "css" %}
```

(continues on next page)

(continued from previous page)

```
    </head>
    <body>
        {% cms_toolbar %}
        {% placeholder "content" %}
        {% render_block "js" %}
    </body>
</html>
```

This is worth explaining in a little detail:

- `{% load cms_tags sekizai_tags %}` loads the template tag libraries we use in the template.
- `{% page_attribute "page_title" %}` extracts the page's `page_title` *attribute*.
- `{% render_block "css" %}` and `{% render_block "js" %}` are Sekizai template tags that load blocks of HTML defined by Django applications. django CMS defines blocks for CSS and JavaScript, and requires these two tags. We recommended placing `{% render_block "css" %}` just before the `</head>` tag, and and `{% render_block "js" %}` tag just before the `</body>`.
- `{% cms_toolbar %}` renders the *django CMS toolbar*.
- `{% placeholder "content" %}` defines a *placeholder*, where plugins can be inserted. A template needs at least one `{% placeholder %}` template tag to be useful for django CMS. The name of the placeholder is simply a descriptive one, for your reference.

Django needs to be know where to look for its templates, so add `templates` to the `TEMPLATES['DIRS']` list:

```
TEMPLATES = [
    {
        ...
        'DIRS': ['templates'],
        ...
    },
]
```

Note: The way we have set up the template here is just for illustration. In a real project, we'd recommend creating a `base.html` template, shared by all the applications in the project, that your django CMS templates can extend.

See Django's template language documentation for more on how template inheritance works.

Media and static file handling

A django CMS site will need to handle:

- *static files*, that are a core part of an application or project, such as its necessary images, CSS or JavaScript
- *media files*, that are uploaded by the site's users or applications.

`STATIC_URL` is defined (as `"/static/"`) in a new project's settings by default. `STATIC_ROOT`, the location that static files will be copied to and served from, is not required for development - only for production.

For now, using the runserver and with `DEBUG = True` in your settings, you don't need to worry about either of these.

However, `MEDIA_URL` (where media files will be served) and `MEDIA_ROOT` (where they will be stored) need to be added to your settings:

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

For deployment, you need to configure suitable media file serving. **For development purposes only**, the following will work in your `urls.py`:

```
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    ...
] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

(See the Django documentation for guidance on serving media files in production.)

Adding content-handling functionality

You now have the basics set up for a django CMS site, which is able to manage and serve up pages. However the project so far has no plugins installed, which means it has no way of handling content in those pages. All content in django CMS is managed via plugins. So, we now need to install some additional addon applications to provide plugins and other functionality.

You don't actually **need** to install any of these. django CMS doesn't commit you to any particular applications for content handling. The ones listed here however provide key functionality and are strongly recommended.

Django Filer

Django Filer provides file and image management. Many other applications also rely on Django Filer - it's very unusual to have a django CMS site that does *not* run Django Filer. The configuration in this section will get you started, but you should refer to the Django Filer documentation for more comprehensive configuration information.

To install:

```
pip install django-filer
```

A number of applications will be installed as dependencies. Easy Thumbnails is required to create new versions of images in different sizes; Django MPTT manages the tree structure of the folders in Django Filer.

Pillow, the Python imaging library, will be installed. Pillow needs some system-level libraries - the Pillow documentation describes in detail what is required to get this running on various operating systems.

Add:

```
'filer',
'easy_thumbnails',
'mptt',
```

to `INSTALLED_APPS`.

You also need to add:

```
THUMBNAIL_HIGH_RESOLUTION = True

THUMBNAIL_PROCESSORS = (
    'easy_thumbnails.processors.colorspace',
```

(continues on next page)

(continued from previous page)

```
    'easy_thumbnails.processors.autocrop',
    'filer.thumbnail_processors.scale_and_crop_with_subject_location',
    'easy_thumbnails.processors.filters'
)
```

New database tables will need to be created for Django Filer and Easy Thumbnails, so run migrations:

```
python manage.py migrate filer
python manage.py migrate easy_thumbnails
```

(or simply, `python manage.py migrate`).

Django CMS CKEditor

Django CMS CKEditor is the default text editor for django CMS.

Install: `pip install djangocms-text-ckeditor`.

Add `djangocms_text_ckeditor` to your `INSTALLED_APPS`.

Run migrations:

```
python manage.py migrate djangocms_text_ckeditor
```

Miscellaneous plugins

There are plugins for django CMS that cover a vast range of functionality. To get started, it's useful to be able to rely on a set of well-maintained plugins that cover some general content management needs.

- djangocms-link
- djangocms-file
- djangocms-picture
- djangocms-video
- djangocms-googlemap
- djangocms-snippet
- djangocms-style
- djangocms-column

To install:

```
pip install djangocms-link djangocms-file djangocms-picture djangocms-video djangocms-
↪googlemap djangocms-snippet djangocms-style djangocms-column
```

and add:

```
'djangocms_link',
'djangocms_file',
'djangocms_picture',
'djangocms_video',
'djangocms_googlemap',
```

(continues on next page)

(continued from previous page)

```
'djangocms_snippet',
'djangocms_style',
'djangocms_column',
```

to `INSTALLED_APPS`.

Then run migrations:

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

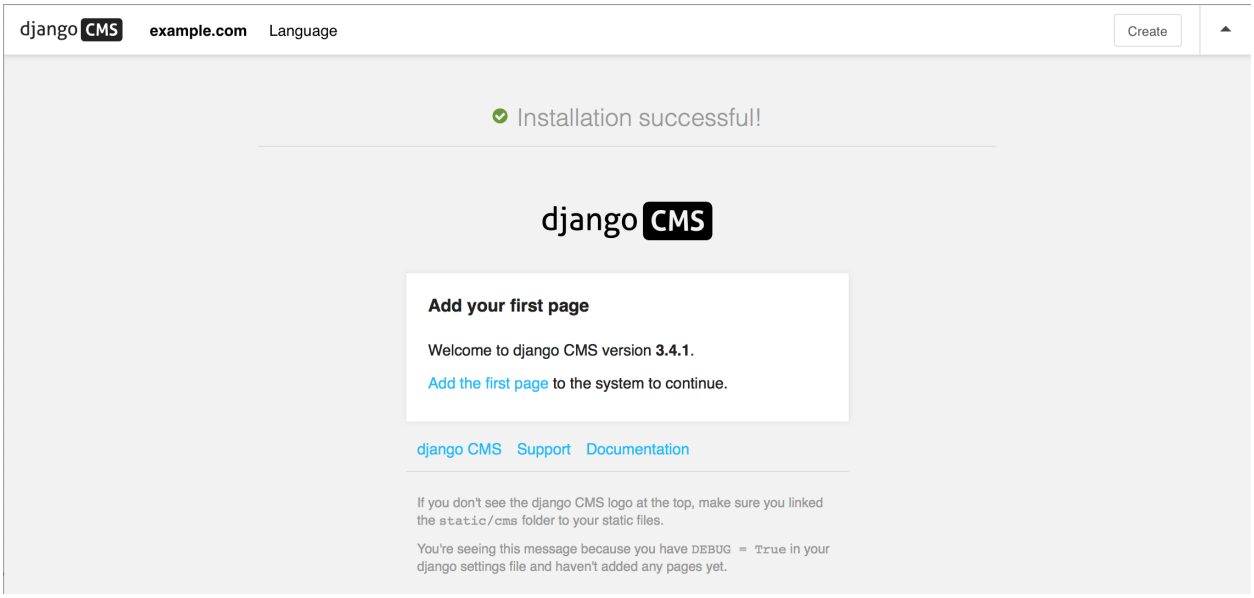
These and other plugins are described in more detail in *Some commonly-used plugins*. More are listed plugins available on the django CMS Marketplace.

Launch the project

Start up the runserver:

```
python manage.py runserver
```

and access the new site, which you should now be able to reach at `http://localhost:8000`. Login if you haven't done so already.



Next steps

If this is your first django CMS project, read through the *Tutorial* for a walk-through of some basics.

The *tutorials for developers* will help you understand how to approach django CMS as a developer. Note that the tutorials assume you have installed the CMS using the django CMS Installer, but with a little adaptation you'll be able to use it as a basis.

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

How to create 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 third-party plugins, you'll have to write your own custom CMS Plugin. Don't worry though - writing a CMS Plugin is very straightforward.

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 sub-class of *cms.models.pluginmodel.CMSPlugin* to **store the configuration** for your plugin instances
- a sub-class 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 sub-class of `django.contrib.admin.ModelAdmin`.

Because *`CMSPluginBase`* sub-classes `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 sub-class 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 _

@plugin_pool.register_plugin
class HelloPlugin(CMSPluginBase):
    model = CMSPlugin
    render_template = "hello_plugin.html"
    cache = False
```

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_name}}{% else %}Guest{% endif %}</h1>
```

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 sub-classes 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.

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 override the *`render()`* method which determines the template context variables that are used to render your plugin. By default, this method only adds `instance` and `placeholder` objects to your context, but plugins can override this to include any context that is required.

A number of other methods are available for overriding on your CMSPluginBase sub-classes. See: *`CMSPluginBase`* for further details.

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 sub-classing *`cms.models.pluginmodel.CMSPlugin`* in the `models.py` of an installed application.

Let's improve our `HelloPlugin` from above by making its fallback name for non-authenticated users configurable.

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

```
from cms.models.pluginmodel import CMSPlugin

from django.db import models

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

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

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

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

from .models import Hello

@plugin_pool.register_plugin
class HelloPlugin(CMSPluginBase):
    model = Hello
    name = _("Hello Plugin")
    render_template = "hello_plugin.html"
    cache = False
```

(continues on next page)

(continued from previous page)

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

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

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

```
<h1>Hello {% if request.user.is_authenticated %}
  {{ request.user.first_name }} {{ request.user.last_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 hard-coded `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 sub-classed models. If you use all core plugins, this includes: `file`, `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

Every time 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):
        # Before copying related objects from the old instance, the ones
        # on the current one need to be deleted. Otherwise, duplicates may
        # appear on the public version of the page
        self.associated_item.all().delete()

        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.

Relations *between* plugins

It is much harder to manage the copying of relations when they are from one plugin to another.

See the GitHub issue copy_relations() does not work for relations between cmsplugins #4143 for more details.

Advanced

Inline Admin

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

```
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.ModelAdmin`, you can customise the form for your plugins just as you would customise 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* sub-class 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.

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"></script>{% endaddtoblock %}
{% addtoblock "js" %}<script type="text/javascript" src="{{ MEDIA_URL }}myplugin/js/
↪myotherfile.js"></script>{% endaddtoblock %}
{% addtoblock "css" %}<link rel="stylesheet" type="text/css" href="{{ MEDIA_URL }}
↪myplugin/css/astylesheet.css">{% endaddtoblock %}
{% 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>
<script type="text/javascript" src="{{ MEDIA_URL }}myplugin/js/myotherfile.js"></
↪script>{% endaddtoblock %}
{% addtoblock "css" %}
    <link rel="stylesheet" type="text/css" href="{{ MEDIA_URL }}myplugin/css/
↪astylesheet.css"></script>
{% 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:

```
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`:

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

In your `yourapp.cms_plugin_processors.py`:

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

@plugin_pool.register_plugin
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
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
```

parent.html:

```
{% load cms_tags %}

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

child.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:

```
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', 'tree_id', 'level',

↪'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)},
            )
        ]

    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'),
        ]
        return urlpatterns
```

(continues on next page)

(continued from previous page)

```
    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 placeholder.")
        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="AliasPlugin")
        if plugin:
            alias.plugin = plugin
        if placeholder:
            alias.alias_placeholder = placeholder
        alias.save()
        return HttpResponse("ok")
```

Plugin data migrations

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

Data migrations are a different story, though.

If your data migration does something like:

```
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 data migration to run before the django CMS migration that creates treebeard fields, by doing this the data migration 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')
        ]
```

How to customise 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 `cms_menus.py` in your application, with the following:

```
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)
```

(continues on next page)

(continued from previous page)

```
        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)
```

Note: Up to version 3.1 this module was named `menu.py`. Please update your existing modules to the new naming convention. Support for the old name will be removed in version 3.5.

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

`title` Text for the menu node

`url` URL for the menu node link

`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 *`menus.base.NavigationNode`* provides a number of methods which are detailed in the *`NavigationNode`* API references.

Customise 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 make use of existing ones.

For example it's possible to add `{'visible_for_anonymous': False}`/`{'visible_for_authenticated': False}` attributes recognised 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': False}),
                NavigationNode(_("Log in"), reverse(login), 3, attr={'visible_for_
↪authenticated': False}),
                NavigationNode(_("Sign up"), reverse(logout), 4, attr={'visible_for_
↪authenticated': False}),
                NavigationNode(_("Log out"), reverse(logout), 2, attr={'visible_for_
↪anonymous': False}),
            ]
```

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.

Example use-cases

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 another example, you might want a particular attribute of your `Pages` to be available in menu templates. In order to keep menu nodes lightweight (which can be important in a site with thousands of pages) they only contain the minimum attributes required to generate a usable menu.

In both cases, a Navigation Modifier is the solution - in the first case, to add a new node at the appropriate place, and in the second, to add a new attribute - on the `attr` attribute, rather than directly on the `NavigationNode`, to help avoid conflicts - to all nodes in the menu.

How it works

Place your modifiers in your application's `cms_menus.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.

Here is an example of a simple modifier that places each Page's `changed_by` attribute in the corresponding `NavigationNode`:

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

from cms.models import Page

class MyExampleModifier(Modifier):
    """
    This modifier makes the changed_by attribute of a page
    accessible for the menu system.
    """
    def modify(self, request, nodes, namespace, root_id, post_cut, breadcrumb):
        # only do something when the menu has already been cut
        if post_cut:
            # only consider nodes that refer to cms pages
            # and put them in a dict for efficient access
            page_nodes = {n.id: n for n in nodes if n.attr["is_page"]}
            # retrieve the attributes of interest from the relevant pages
            pages = Page.objects.filter(id__in=page_nodes.keys()).values('id',
↪'changed_by')
            # loop over all relevant pages
            for page in pages:
                # take the node referring to the page
                node = page_nodes[page['id']]
                # put the changed_by attribute on the node
                node.attr["changed_by"] = page['changed_by']
        return nodes

menu_pool.register_modifier(MyExampleModifier)
```

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 a breadcrumb call rather than a menu call?

Here is an example of a built-in modifier that marks all node levels:

```
class Level(Modifier):
    """
```

(continues on next page)

(continued from previous page)

```
    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)
```

Performance issues in menu modifiers

Navigation modifiers can quickly become a performance bottleneck. Each modifier is called multiple times: For the breadcrumb (`breadcrumb=True`), for the whole menu tree (`post_cut=False`), for the menu tree cut to the visible part (`post_cut=True`) and perhaps for each level of the navigation. Performing inefficient operations inside a navigation modifier can hence lead to big performance issues. Some tips for keeping a modifier implementation fast:

- Specify when exactly the modifier is necessary (in breadcrumb, before or after cut).
- Only consider nodes and pages relevant for the modification.
- Perform as less database queries as possible (i.e. not in a loop).
- In database queries, fetch exactly the attributes you are interested in.
- If you have multiple modifications to do, try to apply them in the same method.

How to create 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.

All URLs in that URL path will be passed to the attached application's URL configs.

The *Tutorials* section contains a basic guide to *getting started with apphooks*. This document assumes more familiarity with the CMS generally.

The basics of apphook creation

To create an apphook, create a `cms_apps.py` file in your application.

The file needs to contain a *CMSApp* sub-class. For example:

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

@apphook_pool.register
class MyApphook(CMSApp):
    name = _("My Apphook")

    def get_urls(self, page=None, language=None, **kwargs):
        return ["myapp.urls"]       # replace this with the path to your application
↪'s URLs module
```

Changed in version 3.3: `CMSApp.get_urls()` replaces `CMSApp.urls`. `urls` is now deprecated and will be removed in version 3.5.

Apphooks for namespaced applications

Does your application use namespaced URLs? This is good practice, so it should!

In that case you will need to ensure that your apphooks include its URLs in the right namespace. Add an `app_name` attribute to the class that reflects the way you'd include the applications' URLs into your project.

For example, if your application requires that your project's URLs do:

```
url(r'^myapp/', include('myapp.urls', app_name='myapp')),
```

then your `MyApphook` class should include:

```
app_name = "myapp"
```

If you fail to this, then any templates in the application that invoke URLs using the form `{% url 'myapp:index' %}` or views that call (for example) `reverse('myapp:index')` will throw a `NoReverseMatch` error.

If you had already assigned a page to your aplication prior to setting the `app_name` attribute, you'll also need to edit its *Advanced settings* and specify your `app_name` in the *Application instance name* field that now appears, to avoid the `NoReverseMatch` error (the instance name is filled automatically in new pages).

Unless the class that defines the apphook specifies an `app_name`, it can be attached only to one page at a time. Attempting to apply it a second time will cause an error. See *Attaching an application multiple times* for more on having multiple apphook instances.

Loading new and re-configured apphooks

Certain apphook-related changes require server restarts in order to be loaded.

Whenever you:

- add or remove an apphook
- change the slug of a page containing an apphook or the slug of a page which has a descendant with an apphook

the URL caches must be reloaded.

If you have the *cms.middleware.utils.ApphookReloadMiddleware* installed, which is recommended, the server will do it for your by re-initialising the URL patterns automatically.

Otherwise, you will need to restart it manually.

Using an apphook

Once your apphook has been set up and loaded, you'll now be able to select the *Application* that's hooked into that page from its *Advanced settings*.

Note: An apphook won't actually do anything until the page it belongs to is published. Take note that this also means all parent pages must also be published.

The apphook attaches all of the apphooked application's URLs to the page; its root URL will be the page's own URL, and any lower-level URLs will be on the same URL path.

So, given an application with the `urls.py`:

```
from django.conf.urls import *

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

attached to a page whose URL path is `/hello/world/`, its views will be exposed as follows:

- `main_view` at `/hello/world/`
- `sample_view` at `/hello/world/sublevel/`

Sub-pages of an apphooked page

Usually, it's simplest to leave an apphook to swallow up all the URLs below its page's URL.

However, as long as the application's urlconf is not too greedy and doesn't conflict with the URLs of any sub-pages, those sub-pages can be reached. That is, although the apphooked application will have priority, any URLs it *doesn't* consume will be available for ordinary django CMS pages, if they exist.

Apphook management

Uninstalling an apphook with applied instances

If you remove an apphook class (in effect uninstalling it) from your system that still has instances applied to pages, django CMS tries to handle this as gracefully as possible:

- Affected Pages still maintain a record of the applied apphook; if the apphook class is reinstated, it will work as before.
- The page list will show apphook indicators where appropriate.
- The page will otherwise behave like a normal django CMS page, and display its placeholders in the usual way.

- If you save the page's Advanced settings, the apphook will be removed.

Management commands

You can clear uninstalled apphook instances using a CMS management command `uninstall apphooks`; for example:

```
manage.py cms uninstall apphooks MyApphook MyOtherApphook
```

You can get a list of installed apphooks using the *cms list*; in this case:

```
manage.py cms list apphooks
```

See the *Management commands reference* for more information.

Apphook menus

Generally, it is recommended to allow the user to control whether a menu is attached to a page. However, an apphook can be made to do this automatically if required. It will behave just as if it were attached the page using its *Advanced settings*).

Menus can be added to an apphook using the `get_menus()` method. On the basis of the example above:

```
# [...]
from myapp.menu import MyAppMenu

class MyApphook(CMSApp):
    # [...]
    def get_menus(self, page=None, language=None, **kwargs):
        return [MyAppMenu]
```

Changed in version 3.3: `CMSApp.get_menus()` replaces `CMSApp.menus`. The `menus` attribute is now deprecated and will be removed in version 3.5.

The menus returned in the `get_menus()` method need to return a list of nodes, in their `get_nodes()` methods. See *Attach Menus* for more on creating menu classes that generate nodes.

You can return multiple menu classes; all will be attached to the same page:

```
def get_menus(self, page=None, language=None, **kwargs):
    return [MyAppMenu, CategoryMenu]
```

Apphook permissions

By default the content represented by an apphook has the same permissions set as the page it is assigned to. So if for example a page requires the user to be logged in, then the attached apphook and all its URLs will have the same requirements.

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

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

If you still want some of your views to use the CMS's permission checks you can enable them via a decorator, `cms.utils.decorators.cms_perms`

Here is a simple example:

```
from cms.utils.decorators import cms_perms

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

If you have your own permission checks in your application, then use `exclude_permissions` property of the apphook:

```
class SampleApp(CMSApp):
    name = _("Sample App")
    permissions = True
    exclude_permissions = ["some_nested_app"]

    def get_urls(self, page=None, language=None, **kwargs):
        return ["project.sampleapp.urls"]
```

For example, django-oscar apphook integration needs to be used with `exclude_permissions` of the dashboard app, because it uses the customisable access function. So, your apphook in this case will look like this:

```
class OscarApp(CMSApp):
    name = _("Oscar")
    exclude_permissions = ['dashboard']

    def get_urls(self, page=None, language=None, **kwargs):
        return application.urls[0]
```

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
- change the slug of a page with a descendant with an apphook

The CMS the server will reload its URL caches. It does this by listening for the signal `cms.signals.urls_need_reloading`.

Warning: This signal does not actually do anything itself. For 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 an API you'll need a request for the signal to fire.

Apphooks and placeholder template tags

It's important to understand that while an apphooked application takes over the CMS page at that location completely, depending on how the application's templates extend other templates, a django CMS `{% placeholder %}` template tag may be invoked - **but will not work**.

`{% static_placeholder %}` tags on the other hand are *not* page-specific and *will* function normally.

How to manage complex apphook configuration

In *How to create apphooks* we discuss some basic points of using apphooks. In this document we will cover some more complex implementation possibilities.

Attaching an application multiple times

Define a namespace at class-level

If you want to attach an application multiple times to different pages, then the class defining the apphook *must* have an `app_name` attribute:

```
class MyApphook(CMSApp):
    name = _("My Apphook")
    app_name = "myapp"

    def get_urls(self, page=None, language=None, **kwargs):
        return ["myapp.urls"]
```

The `app_name` does three key things:

- It provides the *fallback namespace* for views and templates that reverse URLs.
- It exposes the *Application instance name* field in the page admin when applying an apphook.
- It sets the *default apphook instance name* (which you'll see in the *Application instance name* field).

We'll explain these with an example. Let's suppose that your application's views or templates use `reverse('myapp:index')` or `{% url 'myapp:index' %}`.

In this case the namespace of any apphooks you apply must match `myapp`. If they don't, your pages using them will throw up a `NoReverseMatch` error.

You can set the namespace for the instance of the apphook in the *Application instance name* field. However, you'll need to set that to something *different* if an instance with that value already exists. In this case, as long as `app_name = "myapp"` it doesn't matter; even if the system doesn't find a match with the name of the instance it will fall back to the one hard-wired into the class.

In other words setting `app_name` correctly guarantees that URL-reversing will work, because it sets the fallback namespace appropriately.

Set a namespace at instance-level

On the other hand, the *Application instance name* will override the `app_name` *if* a match is found.

This arrangement allows you to use multiple application instances and namespaces if that flexibility is required, while guaranteeing a simple way to make it work when it's not.

Django's Reversing namespaced URLs documentation provides more information on how this works, but the simplified version is:

1. First, it'll try to find a match for the *Application instance name*.
2. If it fails, it will try to find a match for the `app_name`.

Apphook configurations

Namespacing your apphooks also makes it possible to manage additional database-stored apphook configuration, on an instance-by-instance basis.

Basic concepts

To capture the configuration that different instances of an apphook can take, a Django model needs to be created - each apphook instance will be an instance of that model, and administered through the Django admin in the usual way.

Once set up, an apphook configuration can be applied to to an apphook instance, in the *Advanced settings* of the page the apphook instance belongs to:



The configuration is then loaded in the application's views for that namespace, and will be used to determined how it behaves.

Creating an application configuration in fact creates an apphook instance namespace. Once created, the namespace of a configuration cannot be changed - if a different namespace is required, a new configuration will also need to be created.

An example apphook configuration

In order to illustrate how this all works, we'll create a new FAQ application, that provides a simple list of questions and answers, together with an apphook class and an apphook configuration model that allows it to exist in multiple places on the site in multiple configurations.

We'll assume that you have a working django CMS project running already.

Using helper applications

We'll use a couple of simple helper applications for this example, just to make our work easier.

Aldryn Apphooks Config

Aldryn Apphooks Config is a helper application that makes it easier to develop configurable apphooks. For example, it provides an `AppHookConfig` for you to subclass, and other useful components to save you time.

In this example, we'll use Aldryn Apphooks Config, as we recommend it. However, you don't have to use it in your own projects; if you prefer to can build the code you require by hand.

Use `pip install aldryn-apphooks-config` to install it.

Aldryn Apphooks Config in turn installs Django AppData, which provides an elegant way for an application to extend another; we'll make use of this too.

Create the new FAQ application

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

Create the FAQ `Entry` model

`models.py`:

```
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 a `ForeignKey` to an apphook configuration model; we'll create it in a moment. This model will hold the specific namespace configuration, and makes it possible to assign each FAQ Entry to a namespace.

The custom `AppHookConfigManager` is there to make it easy to filter the queryset of `Entries` using a convenient shortcut: `Entry.objects.namespace('foobar')`.

Define the AppHookConfig subclass

In a new file `cms_appconfig.py` in the FAQ application:

```
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'),
```

(continues on next page)

(continued from previous page)

```
        blank=False,
        default=5,
    )


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

The implementation *can* be left completely empty, as the minimal schema is already defined in the abstract parent model provided by Aldryn Apphooks Config.

Here though we're defining an extra field on model, `paginate_by`. We'll use it later to control how many entries should be displayed per page.

We also set up a `FaqConfigForm`, which uses `AppDataForm` to add a field to `FaqConfig` without actually touching its model.

The title field could also just be a model field, like `paginate_by`. But we're using the AppDataForm to demonstrate this capability.

Define its admin properties

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

```
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.`.

Define the apphook itself

Now let's create the apphook, and set it up with support for multiple instances. In `cms_apps.py`:

```
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


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

Define a list view for FAQ entries

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. In `views.py`:

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


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` saves you the work of setting any attributes in your view - it automatically sets, for the view class instance:

- current namespace in `self.namespace`
- namespace configuration (the instance of FaqConfig) 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`. `qs.namespace`, thanks to the model manager we defined earlier, is the equivalent of `qs.filter(app_config__namespace=self.namespace)`.

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

Define a template

In `faq/templates/faq/index.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>
        </div>
    {% endif %}
{% endblock %}
```

URLconf

`urls.py`:

```
from django.conf.urls import url
from . import views


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

Put it all together

Finally, we add `faq` to `INSTALLED_APPS`, then create and run migrations:

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

Now we should be all set.

Create two pages with the `faq` apphook (don't forget to publish them), with different namespaces and different configurations. Also create some entries assigned to the two namespaces.

You can experiment with the different configured behaviours (in this case, only pagination is available), and the way that different `Entry` instances can be associated with a specific apphook.

How to serve multiple languages

If you used the django CMS installer to start your project, you'll find that it's already set up for serving multilingual content. Our *How to install django CMS by hand* guide also does the same.

This guide specifically describes the steps required to enable multilingual support, in case you need to it manually.

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 a full example of `urls.py`:

```
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')),
)
```

Monolingual URLs

Of course, if you want only monolingual URLs, without a language code, simply don't use `i18n_patterns()`:

```
urlpatterns += [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^', include('cms.urls')),
]
```

Store the user's language preference

The user's preferred language is maintained through a browsing session. So that django CMS remembers the user's preference in subsequent sessions, it must be stored in a cookie. To enable this, `cms.middleware.language.LanguageCookieMiddleware` must be added to the project's `MIDDLEWARE_CLASSES` setting.

See *How django CMS determines which language to serve* for more information about how this works.

Working in templates

Display a language chooser in the page

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`

Get the URL of the current page for a different language

The `page_language_url` returns the URL of the current page in another language.

Example:

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

Configuring language-handling behaviour

CMS_LANGUAGES describes the all options available for determining how django CMS serves content across multiple languages.

How to work with templates

Application can reuse cms templates by mixing cms template tags 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 template tag 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 memorises 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.

How to extend Page & Title models

You can extend the *cms.models.Page* and *cms.models.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.

Title vs Page extensions

The difference between a **page extension** and a **title extension** is related to the difference between the *cms.models.Page* and *cms.models.Title* models.

- `PageExtension`: use to add fields that should have **the same values** for the different language versions of a page - for example, an icon.
- `TitleExtension`: use to add fields that should have **language-specific values** for different language versions of a page - for example, keywords.

Implement a basic extension

Three basic steps are required:

- add the extension *model*
- add the extension *admin*
- add a toolbar menu item for the extension

Page model extension example

The model

To add a field to the Page model, create a class that inherits from `cms.extensions.PageExtension`. Your class should live in one of your applications' `models.py` (or module).

Note: 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 using `extension_pool`.

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

```
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)
```

Of course, you will need to make and run a migration for this new model.

The admin

To make your extension editable, you must first create an admin class that sub-classes `cms.extensions.PageExtensionAdmin`. This admin handles page permissions.

Note: 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:

```
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 appropriate to your needs.

Note: Note that the field that holds the relationship between the extension and a CMS Page is non-editable, so it does not appear directly in the Page admin views. This may be addressed in a future update, but in the meantime the toolbar provides access to it.

The toolbar item

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.

To add toolbar items for your extension create a file named `cms_toolbars.py` in one of your apps, and add the relevant menu entries for the extension on each page.

Here's a simple version for our example. This example adds a node to the existing *Page* menu, called *Page icon*. When selected, it will open a modal dialog in which the *Page icon* field can be edited.

```
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):
    # defines the model for the current toolbar
    model = IconExtension

```

(continues on next page)

(continued from previous page)

```
    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 in position 0 (at the top of the menu)
                current_page_menu.add_modal_item(_('Page Icon'), url=url,
                    disabled=not self.toolbar.edit_mode, position=0)
```

Title model extension example

In this example, we'll create a `Rating` extension field, that can be applied to each `Title`, in other words, to each language version of each `Page`.

Note: Please refer to the more detailed discussion above of the Page model extension example, and in particular to the special **notes**.

The model

```
from django.db import models
from cms.extensions import TitleExtension
from cms.extensions.extension_pool import extension_pool


class RatingExtension(TitleExtension):
    rating = models.IntegerField()


extension_pool.register(RatingExtension)
```

The admin

```
from django.contrib import admin
from cms.extensions import TitleExtensionAdmin
from .models import RatingExtension


class RatingExtensionAdmin(TitleExtensionAdmin):
    pass


admin.site.register(RatingExtension, RatingExtensionAdmin)
```

The toolbar item

In this example, we need to loop over the titles for the page, and populate the menu with those.

```
from cms.toolbar_pool import toolbar_pool
from cms.extensions.toolbar import ExtensionToolbar
from django.utils.translation import ugettext_lazy as _
from .models import RatingExtension
from cms.utils import get_language_list  # needed to get the page's languages
@toolbar_pool.register
class RatingExtensionToolbar(ExtensionToolbar):
    # defines the model for the current toolbar
    model = RatingExtension

    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 labelled "Ratings" at position 1 in the menu
            sub_menu = self._get_sub_menu(
                current_page_menu, 'submenu_label', 'Ratings', position=1
                )

            # retrieves the instances of the current title extension (if any)
            # and the toolbar item URL
            urls = self.get_title_extension_admin()

            # we now also need to get the titleset (i.e. different language titles)
            # for this page
            page = self._get_page()
            titleset = page.title_set.filter(language__in=get_language_list(page.site_
↪id))

            # create a 3-tuple of (title_extension, url, title)
            nodes = [(title_extension, url, title.title) for (
                (title_extension, url), title) in zip(urls, titleset)
                ]

            # cycle through the list of nodes
            for title_extension, url, title in nodes:

                # adds toolbar items
                sub_menu.add_modal_item(
                    'Rate %s' % title, url=url, disabled=not self.toolbar.edit_mode
                    )
```

Using extensions

In templates

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

Page extensions

As per the normal related_name naming mechanism, the appropriate field to access is the same as your `PageExtension` 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.

Title extensions

In order to retrieve a title extension within a template, get the `Title` object using `request.current_page.get_title_obj`. Using the example above, we could use:

```
{{ request.current_page.get_title_obj.ratingextension.rating }}
```

With menus

Like most other Page attributes, extensions are not represented in the menu `NavigationNodes`, and therefore menu templates will not have access to them by default.

In order to make the extension accessible, you'll need to create a *menu modifier* (see the example provided) that does this.

Each page extension instance has a one-to-one relationship with its page. Get the extension by using the reverse relation, along the lines of `extension = page.yourextensionlowercased`, and place this attribute of `page` on the node - as (for example) `node.extension`.

In the menu template the icon extension we created above would therefore be available as `child.extension.icon`.

Handling relations

If your `PageExtension` or `TitleExtension` includes a ForeignKey *from* another model or includes a ManyToManyField, 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 `ManyToManyField`

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


class MyPageExtension(PageExtension):

    page_categories = models.ManyToManyField(Category, blank=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)
```

Complete toolbar API

The example above uses the *Simplified 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:

```
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.page_permissions import user_can_change_page
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

        if user_can_change_page(user=self.request.user, page=self.page):
            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.page.pk
```

(continues on next page)

(continued from previous page)

```
            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 publish the page.

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;
- *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;
- *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;
- *cms.toolbar.toolbar.CMSToolbar.get_or_create_menu()*
- *cms.extensions.toolbar.ExtensionToolbar._setup_extension_toolbar()*

How to extend the Toolbar

New in version 3.0.

You can add and remove toolbar items. This allows you to integrate django CMS's frontend editing mode into your application, and provide your users with a streamlined editing experience.

For the toolbar API reference, please refer to *The Toolbar*.

Important: Overlay and **sideframe**

Then django CMS *sideframe* has been replaced with an *overlay* mechanism. The API still refers to the `sideframe`, because it is invoked in the same way, and what has changed is merely the behaviour in the user's browser.

In other words, *sideframe* and the *overlay* refer to different versions of the same thing.

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

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

These classes 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 you to 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 can 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.

`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:

```
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 contain different items 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_menu()*.

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:

```
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_toolbars 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 *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 aid developers, django-cms exposes a *cms.toolbar.items.ToolbarMixin.get_alphabetical_insert_position()* method, which, if used consistently, can produce alphabetised 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_toolbars import ADMIN_MENU_IDENTIFIER, ADMINISTRATION_BREAK

@toolbar_pool.register
class OfficesToolbar(CMSToolbar):

    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.
```

(continues on next page)

(continued from previous page)

```
        #
        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)

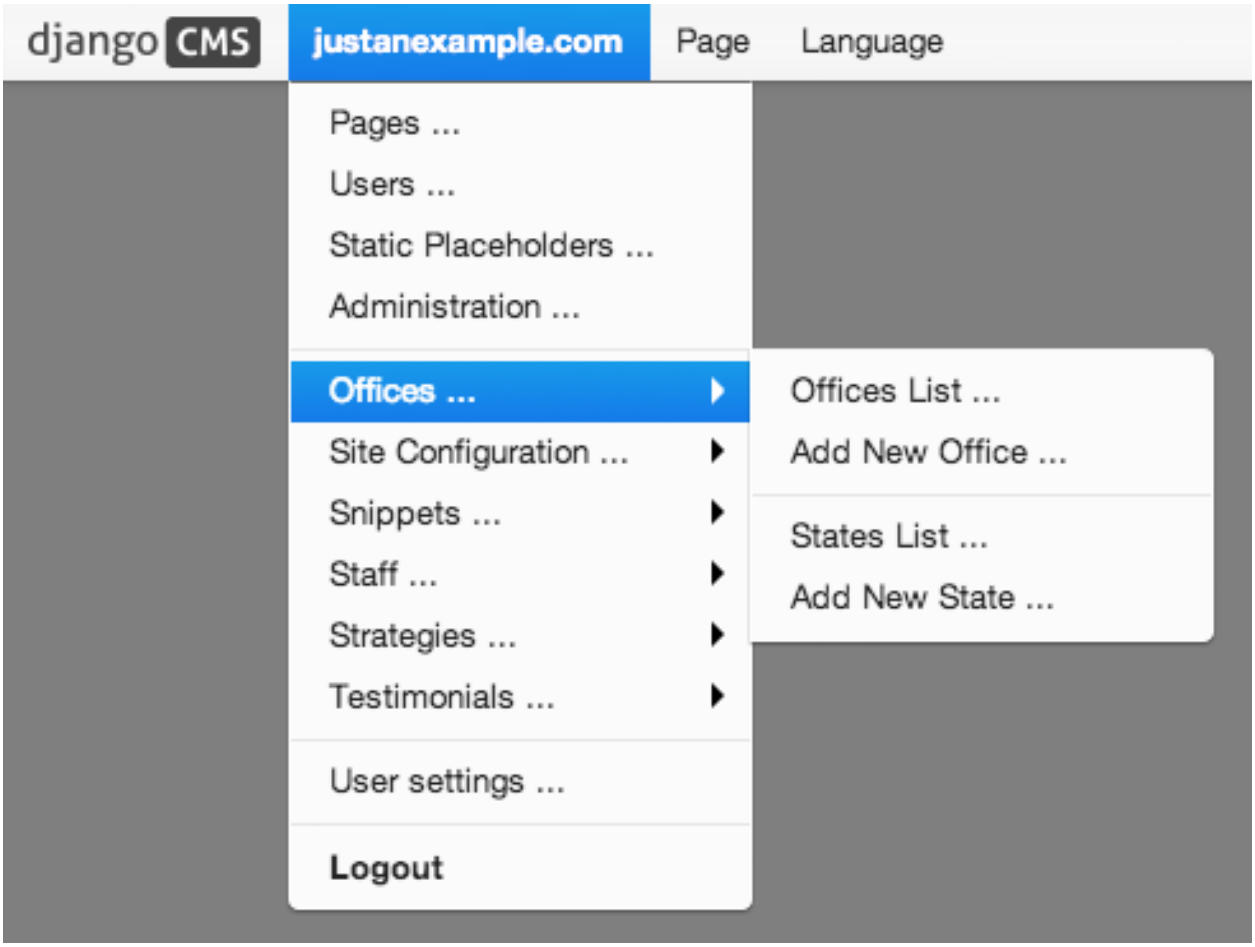
        # Display a modal dialogue for creating a new office...
        url = reverse('admin:offices_office_add')
        office_menu.add_modal_item(_('Add New Office'), url=url)

        # Add a break in the sub-menus
        office_menu.add_break()

        # More sub-menus...
        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:

```
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_id]))
    menu.add_sideframe_item(_('Show History of this Poll'), url=reverse('admin:polls_
↪poll_history', args=[poll_id]))
    menu.add_sideframe_item(_('Delete this Poll'), url=reverse('admin:polls_poll_
↪delete', args=[poll_id]))

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

Detecting URL changes

Sometimes toolbar entries allow you to change the URL of the current object displayed in the website.

For example, suppose you are viewing a blog entry, and the toolbar allows the blog slug or URL to be edited. 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 behaviour manually you can set the `request.toolbar.set_object()` function on which you can set the current object.

Example:

```
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:

```
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 fully expanded. We also add `cms-toolbar-expanding` and `cms-toolbar-collapsing` classes while toolbar is animating.

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 () { ... });
```

How to test 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:

```
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:

```
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:

```
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:

```
from django.test import TestCase
from django.test.client import RequestFactory

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

from myapp.cms_plugins import MyPlugin
```

(continues on next page)

(continued from previous page)

```
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',
        )
        renderer = ContentRenderer(request=RequestFactory())
        html = renderer.render_plugin(model_instance, {})
        self.assertEqual(html, '<strong>Test</strong>')
```

How to use 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*.

Get started

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

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

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

The *PlaceholderField* 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:

```
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:

```
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`:

```
from django.shortcuts import get_object_or_404, render

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

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. Required permissions for edit actions are:

- to add: require add **or** change permission on related Model or instance.
- to change: require add **or** change permission on related Model or instance.
- to delete: require add **or** change **or** delete permission on related Model or instance.

With this logic, an user who can change a Model's instance but can not add a new Model's instance will be able to add some placeholders or plugins to existing Model's instances.

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 authentication 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:

```
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:
```

(continues on next page)

(continued from previous page)

```
            return True
    return False
```

How to manage caching

Set-up

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:

```
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 behaviours:

- *CMS_PAGE_CACHE*
- *CMS_PLACEHOLDER_CACHE*

- *CMS_PLUGIN_CACHE*

How to enable 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 change form for that model.

Note: This interface is not currently available for touch-screen users, but will be improved in future releases.

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

Template tags

This feature relies on five template tags sharing common code. All require that you `{% load cms_tags %}` in your template:

- *render_model* (for editing a specific field)
- *render_model_block* (for editing any of the fields in a defined block)
- *render_model_icon* (for editing a field represented by another value, such as an image)
- *render_model_add* (for adding an instance of the specified model)
- *render_model_add_block* (for adding an instance of the specified model)

Look at the tag-specific page for more detailed reference and discussion of limitations and caveats.

Page titles edit

For CMS pages you can edit the titles from the frontend; according to the attribute specified a default field, which can also be overridden, 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 template tags:

```
{% 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:

```
<template class="cms-plugin cms-plugin-start cms-plugin-cms-page-changelist-1"></
↪tempate>
    <h3>Menu</h3>
    <ul>
        <li><a href="/">Home</a></li>
        <li><a href="/another">another</a></li>
        [...]
<template class="cms-plugin cms-plugin-end cms-plugin-cms-page-changelist-1"></
↪tempate>
```

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):

```
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` template tag:

```
{% load cms_tags %}

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

See *template tag 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:

```
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 template tag:

```
{% 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 template tag 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 template tag 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 *template tag reference* for arguments documentation.

Example `view_url`:

```
{% load cms_tags %}

{% block content %}
<h1>{% render_model instance "some_attribute" "some_field,other_field" ""
↪"admin:exampleapp_example1_some_view" %}</h1>
{% 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 template tag, add quoted sequence of filters as in Django `filter` template tag:

```
{% load cms_tags %}

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

Context variable

The template tag 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 %}
```

How to create sitemaps

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 *cms.sitemaps.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 `from django.contrib.sitemaps.views import sitemap` to `urls.py`\`
- add `url(r'^sitemap\.xml$', sitemap, {'sitemaps': {'cmspages': CMSSitemap}}),` to your `urlpatterns`

`django.contrib.sitemaps`

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

New in version 3.0.

How to manage Page Types

Page Types make it easier for content editors to create pages from predefined **types**.

The examples contain content such as plugins that will be copied over to the newly-created page, leaving the type untouched.

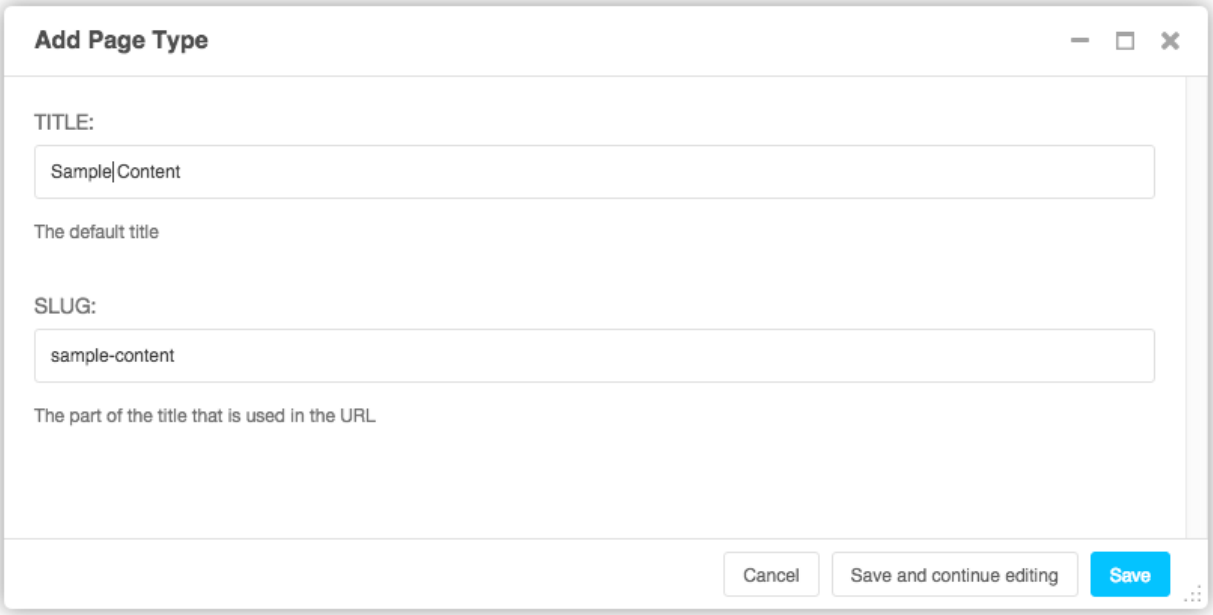
Creating Page Types

First you need to create a new page in the usual way; this will become the template for your new page type.

Use this page as your template to add example content and plugins until you reach a satisfied result.

Once ready, choose *Save as Page Type...* from the *Page* menu and give it an appropriate name. Don't worry about making it perfect, you can continue to change its content and settings.

This will create a new page type, and makes it available from *Add Page* command and the **Create** wizard dialog.



If you don't want or need the original page from which you create the new page type, you can simply delete it.

Managing Page Types

When you save a page as a page type, it is placed in the page list under *Page Types* node.

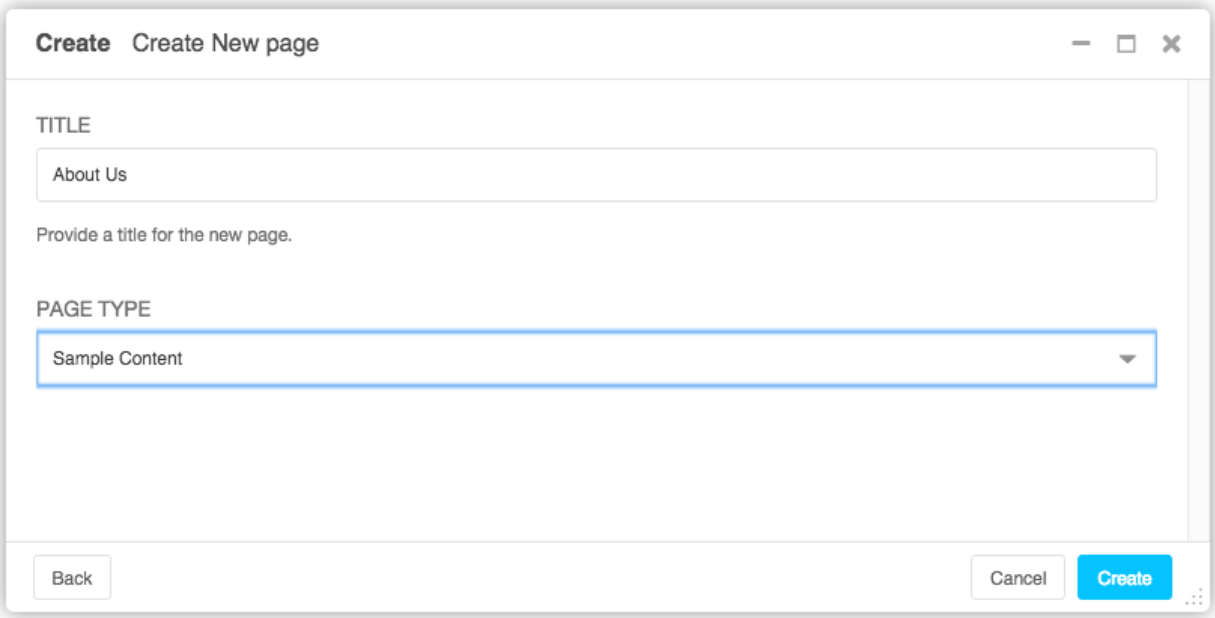
This node behaves differently from regular pages:

- They are not publicly accessible.
- All pages listed in *Page Types* will be rendered in the *Page Types* drop-down menu.

There's also a quick way to create a new page type: simply drag an existing page to the *Page Types* node, whereupon it will become a new page type.

Selecting a Page Type

You can now select a page type when creating a new page. You'll find a drop-down menu named *Page Type* from which you can select the type for your new page.



New in version 3.2.

How to implement content creation wizards

django CMS offers a framework for creating 'wizards' - helpers - for content editors.

They provide a simplified workflow for common tasks.

A django CMS Page wizard already exists, but you can create your own for other content types very easily.

Create a content-creation wizard

Creating a CMS content creation wizard for your own module is fairly easy.

To begin, create a file in the root level of your module called `forms.py` to create your form(s):

```
# my_apps/forms.py

from django import forms

class MyAppWizardForm(forms.ModelForm):
    class Meta:
        model = MyApp
        exclude = []
```

Now create another file in the root level called `cms_wizards.py`. In this file, import `Wizard` as follows:

```
from cms.wizards.wizard_base import Wizard
from cms.wizards.wizard_pool import wizard_pool
```

Then, simply subclass `Wizard`, instantiate it, then register it. If you were to do this for `MyApp`, it might look like this:

```
# my_apps/cms_wizards.py

from cms.wizards.wizard_base import Wizard
from cms.wizards.wizard_pool import wizard_pool

from .forms import MyAppWizardForm

class MyAppWizard(Wizard):
    pass

my_app_wizard = MyAppWizard(
    title="New MyApp",
    weight=200,
    form=MyAppWizardForm,
    description="Create a new MyApp instance",
)

wizard_pool.register(my_app_wizard)
```

Note: If your model doesn't define a `get_absolute_url` function then your wizard will require a *get_success_url* method.

```
class MyAppWizard(Wizard):

    def get_success_url(self, obj, **kwargs):
        """
        This should return the URL of the created object, «obj».
        """
        if 'language' in kwargs:
            with force_language(kwargs['language']):
                url = obj.get_absolute_url()
        else:
            url = obj.get_absolute_url()

        return url
```

That's it!

Note: The module name `cms_wizards` is special, in that any such-named modules in your project's Python path will automatically be loaded, triggering the registration of any wizards found in them. Wizards may be declared and registered in other modules, but they might not be automatically loaded.

The above example is using a `ModelForm`, but you can also use `forms.Form`. In this case, you **must** provide the model class as another keyword argument when you instantiate the Wizard object.

For example:

```
# my_apps/forms.py

from django import forms

class MyAppWizardForm(forms.Form):
    name = forms.CharField()
```

(continues on next page)

(continued from previous page)

```
# my_apps/cms_wizards.py

from cms.wizards.wizard_base import Wizard
from cms.wizards.wizard_pool import wizard_pool

from .forms import MyAppWizardForm
from .models import MyApp

class MyAppWizard(Wizard):
    pass

my_app_wizard = MyAppWizard(
    title="New MyApp",
    weight=200,
    form=MyAppWizardForm,
    model=MyApp,
    description="Create a new MyApp instance",
)

wizard_pool.register(my_app_wizard)
```

You must subclass `cms.wizards.wizard_base.Wizard` to use it. This is because each wizard's uniqueness is determined by its class and module name.

See the *Reference section on wizards* for technical details of the wizards API.

How to contribute a patch

Note: For more background on the material covered in this how-to section, see the *Contributing code* and *Running and writing tests* sections of the documentation.

---

django CMS is an open project, and welcomes the participation of anyone who would like to contribute, whatever their any level of knowledge.

As well as code, we welcome contributions to django CMS's *documentation* and *translations*.

Note: Feel free to dive into coding for django CMS in whichever way suits you. However, you need to be aware of the *guidelines* and *policies* for django CMS project development. Adhering to them will make much easier for the core developers to validate and accept your contribution.

The basics

The basic workflow for a code contribution will typically run as follows:

1. Fork the django CMS project GitHub repository to your own GitHub account
2. Clone your fork locally:

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

3. Create a virtualenv:

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

4. Install its dependencies:

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

   Replace `X.Y` with whichever version of Django you want to work with.

5. Create a new branch for your work:

```
git checkout -b my_fix
```

6. Edit the django CMS codebase to implement the fix or feature.
7. Run the test suite:

```
python manage.py test
```

8. Commit and push your code:

```
git commit
git push origin my_fix
```

9. Open a pull request on GitHub.

Target branches

See *Branches* for information about branch policy.

How to write a test

The 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. We'll assume you have set up your environment correctly as *described above*.

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

```
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 returned `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):

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

Let's try to run it:

```
python manage.py test cms.tests.test_plugins.SimplePluginTestCase
```

This will call the new test case class only and it's handy when creating new tests and iterating quickly through the steps. A full test run (`python manage.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 an empty one:

```
class SimplePluginTestCase(CMSTestCase):

    def test_render_method(self):
        pass
```

Running the test command again will return a slightly 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:

```
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
```

The output is quite similar to the previous run, but 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 utilities provided by django CMS. Before attempting to write a test, take your time to explore the content of `cms.test_utils` package 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 noted above, always create a new branch for your code, be it a fix or a new feature, before committing changes, then create your pull request from your branch to the *target branch* on django CMS.

Acceptance criteria

Matching these criteria from the very beginning will help the core developers to be able to review your submission more quickly and efficiently and will increase the chances of making a successful pull request.

Please see our *Development policies* for guidance on which branches to use, how to prepare pull requests and so on.

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 behaviour
- 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 behaviour
- 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.

4.1.3 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*.

Application hooks ("apphooks")

An Application Hook, usually simply referred to as an apphook, is a way of attaching the functionality of some other application to a django CMS page. It's a convenient way of integrating other applications into a django CMS site.

For example, suppose you have an application that maintains and publishes information about Olympic records. You could add this application to your site's `urls.py` (before the django CMS URLs pattern), so that users will find it at `/records`.

However, although it would thus be integrated into your *project*, it would not be fully integrated into django CMS, for example:

- django CMS would not be aware of it, and - for example - would allow your users to create a CMS page with the same `/records` slug, that could never be reached.
- The application's pages won't automatically appear in your site's menus.
- The application's pages won't be able to take advantage of the CMS's publishing workflow, permissions or other functionality.

Apphooks offer a more complete way of integrating other applications, by attaching them to a CMS page. In this case, the attached application takes over the page and its URL (and all the URLs below it, such as `/records/1984`).

The application can be served at a URL defined by the content managers, and easily moved around in the site structure.

The *Advanced settings* of a CMS page provides an *Application* field. *Adding an apphook class* to the application will allow it to be selected in this field.

Multiple apphooks per application

It's possible for an application to be added multiple times, to different pages. See *Attaching an application multiple times* for more.

Also possible to provide **multiple apphook configurations**:

Apphook configurations

You may require the same application to behave differently in different locations on your site. For example, the Olympic Records application may be required to publish athletics results at one location, but cycling results at another, and so on.

An *apphook configuration* class allows the site editors to create multiple configuration instances that specify the behaviour. The kind of configuration available is presented in an admin form, and determined by the application developer.

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.

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.

Internationalisation

django CMS excels in its multilingual support, and can be configured to handle a vast range of different requirements. Its behaviour is flexible and can be controlled at a granular level in `CMS_LANGUAGES`. Other *Internationalisation and localisation (I18N and L10N)* settings offer further control.

See *How to serve multiple languages* on how to set up a multilingual django CMS project.

URLs

Multilingual URLs require the use of `i18n_patterns()`. For more information about this see the official Django documentation on the subject. *Multilingual URLs* describes what you need to do in a django CMS project.

How django CMS determines which language to serve

django CMS uses a number of standard Django mechanisms to choose the language for the user, in the following order of preference:

- language code in the URL - for example, `http://example.com/de` (when multilingual URLs are enabled)
- language stored in the browsing session
- language stored in a cookie from a previous session
- language requested by the browser in the `Accept-Language` header

- the default `LANGUAGE_CODE` in the site's settings

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

Permissions

The django CMS permissions system is flexible, granular and multi-layered - it can however also sometimes seem a little confusing!

It's important to understand its parts and how they interact in order to use it effectively.

In django CMS permissions can be granted:

- that determine **what actions a user may perform**
- that determine **on which parts of the site they may perform them**

These two dimensions of permissions are independent of each other. See *Permission strategies* below.

`CMS_PERMISSION` mode

The first thing to understand is that as far as permissions are concerned django CMS operates in one of two modes, depending on the *`CMS_PERMISSION`* setting:

- `False` (the default): only the standard Django Users and Groups permissions will apply
- `True`: as well as standard Django permissions, django CMS applies a layer of permissions control affecting *pages*

`CMS_PERMISSION` mode *off*

When django CMS's own permission system is disabled, you have no control over permissions over particular pages. In other words, *row-level controls on pages* do not exist.

You still have control over **what functionality** particular users and groups have in the CMS, but not over **which content** they can exercise it on.

Key user permissions

You can find the permissions you can set for a user or groups in the Django admin, in the *Authentication and Authorization* section.

Filtering by `cms` will show the ones that belong to the CMS application. Permissions that a CMS editor will need are likely to include:

- `cms | cms plugin`
- `cms | page`
- `cms | placeholder`
- `cms | placeholder reference`
- `cms | static placeholder`
- `cms | placeholder reference`
- `cms | title`

Most of these offer the usual add/change/delete options, though there are some exceptions, such as `cms | placeholder | Can use Structure mode`.

Users with permission to do something to a CMS model will be able to do it to *all* instances of that model when `CMS_PERMISSION` mode is *off*

`CMS_PERMISSION` mode *on*

When django CMS's permission system is enabled, a new layer of permissions is **added**, and permissions over CMS page-related models will need to be provided **in addition** to those granted in Django's *Authentication and Authorization* models.

In other words, **both** Django's and django CMS's permissions will need to be granted over pages if an editor is to have access to them.

By default, when `CMS_PERMISSION` mode is enabled, users will not be able to edit CMS pages unless they are Django superusers. This is rarely desirable, so you will probably wish to configure the CMS permissions to provide more nuanced control.

See *Page permissions* below for more.

New admin models

When `CMS_PERMISSION` is enabled, you'll find three new models available in the admin:

- *Pages global permissions*
- *User groups (page)*
- *Users (page)*

You will find that the latter two simply reflect the Django Groups and User permissions that already exist in the system. They are a simpler representation of the available permissions, specific to page editing. You'll often find it more useful to use the Django Groups and User permissions.

Pages global permissions are described below.

Page permissions

When `CMS_PERMISSION` is enabled, unless you simply make your users superusers, you'll need to give each one either global permission, or permission over specific pages (*preferably via their membership of a group* in either case).

Both global and specific permission granting are described below.

Global page permissions

Pages global permissions are available in the admin, in the *django CMS* section.

The first two options for a global permission concern **whom** they apply to.

Then there is list of **what actions** the editor can perform. The editors will need at least *some* of these if they are to manage pages.

Finally, there's a list of the **sites** they can perform the actions on.

Page-specific permissions

The CMS permissions system also provides permisions control for particular pages or hierarchies of pages in the site - row-level permissions, in other words.

These are controlled by selecting *Permissions* from the *Page* menu in the toolbar when on the page (this options is only available when `CMS_PERMISSION` mode is on).

Login required determines whether anonymous visitors will be able to see the page at all.

Menu visibility determines who'll be able to see the page in navigation menus - everyone, or logged in or anonymous users only.

View restrictions determine which groups and users will be able to see the page. Adding a view restriction will allow you to set this. Note that this doesn't apply new restrictions to users who are also editors with appropriate permissions.

Page permissions determine what editors can do to a page (or hierarchy of pages). They work just like the *Pages global permissions* described above, but don't apply globally. They are **added to** global permissions - they don't override them.

The *Can change permission* refers to whether the user can change the permissions of a "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

  Even though a user may have permissions to change a page, that doesn't give them permissions to add or change plugins *within* that page. In order to be able to add/change/delete plugins on any page, you will need to go through the standard Django permissions to provide users with the actions they can perform.

  Even if a *page permission* allows a user to edit pages in general (global) or a particular page (specific), they will still need `cms | page | Can publish page` permission to publish it, `cms | cms plugins | Can edit cms plugin` to edit plugins on the page, and so on.

  This is because the page permissions system is an additional layer over the Django permissions system.

Permission strategies

For a simple site with only a few users you may not need to be concerned about this, but with thousands of pages belonging to different departments and users with greatly differing levels of authority and expertise, it is important to understand who is able to do what on your site.

Two dimensions of permissions

As noted earlier, it's useful to think of your users' permissions across two dimensions:

- what sort of things this user or group of user should be allowed to do (e.g. publish pages, add new plugins, create new users, etc)
- which sections of the site the user should be allowed to do them on (the home page, a limited set of departmental pages, etc)

Use permissions on Groups, not on Users

Avoid applying permissions to individual users unless strictly necessary. It's far better to apply them to Groups, and add Users to Groups. Otherwise, you risk ending up with large numbers of Users with unknown or inappropriate permissions.

Use Groups to build up permissions

Different users may require different subsets of permissions. For example, you could define a *Basic content editor* group, who can edit and publish pages and content, but who don't have permission to create new ones; that permission would be granted to a *Lead content editor* Group. Another Group could have permissions to use the weblog.

Some users should be allowed to edit some pages but not others. So, you could create a *Pharmacy department* and a *Neurology department* group, which don't actually have any permissions of their own, but give each one *Page-specific permissions* on the appropriate landing page of the website.

Then, when managing a user, place the user into the appropriate groups.

Global or specific page permissions?

In a simple site, if you have `CMS_PERMISSION` enabled, add a global permission so that all editors can edit all pages.

If you need more control, only allow select users access to the global permission, but add specific page permissions to pages as appropriate for the other editors.

Using touch-screen devices with django CMS

Important: These notes about touch interface support apply only to the **django CMS admin and editing interfaces**. The visitor-facing published site is **wholly independent** of this, and the responsibility of the site developer.

General

django CMS has made extensive use of double-click functionality, which lacks an exact equivalent in touch-screen interfaces. The touch interface will interpret taps and touches in an intelligent way.

Depending on the context, a tap will be interpreted to mean *open for editing* (that is, the equivalent of a double-click), or to mean *select* (the equivalent of a single click), according to what makes sense in that context.

Similarly, in some contexts similar interactions may *drag* objects, or may *scroll* them, depending on what makes most sense. Sometimes, the two behaviours will be present in the same view, for example in the page list, where certain areas are draggable (for page re-ordering) while other parts of the page can be used for scrolling.

In general, the chosen behaviour is reasonable for a particular object, context or portion of the screen, and in practice is quicker and easier to apprehend simply by using it than it is to explain.

Pop-up help text will refer to clicking or tapping depending on the device being used.

Be aware that some hover-related user hints are simply not available to touch interface users. For example, the overlay (formerly, the *sideframe*) can be adjusted for width by dragging its edge, but this is not indicated in a touch-screen interface.

Device support

Smaller devices such as most phones are too small to be adequately usable. For example, your Apple Watch is sadly unlikely to provide a very good django CMS editing experience.

Older devices will often lack the performance to support a usefully responsive frontend editing/administration interface.

The following devices are known to work well, so newer devices and more powerful models should also be suitable:

- iOS: Apple iPad Air 1, Mini 4
- Android: Sony Xperia Z2 Tablet, Samsung Galaxy Tab 4
- Windows 10: Microsoft Surface

We welcome feedback about specific devices.

Your site's frontend

django CMS's toolbar and frontend editing architecture rely on good practices in your own frontend code. To work well with django CMS's responsive management framework, your own site should be friendly towards multiple devices.

Whether you use your own frontend code or a framework such as Bootstrap 3 or Foundation, be aware that problems in your CSS or markup can affect django CMS editing modes, and this will become especially apparent to users of mobile/hand-held devices.

Known issues

General issues

- Editing links that lack sufficient padding is currently difficult or impossible using touch-screens.
- Similarly, other areas of a page where the visible content is composed entirely of links with minimal padding around them can be difficult or impossible to open for editing by tapping. This can affect the navigation menu (double-clicking on the navigation menu opens the page list).
- Adding links is known to be problematic on some Android devices, because of the behaviour of the keyboard.
- On some devices, managing django CMS in the browser's *private* (also known as *incognito*) mode can have significant performance implications.

  This is because local storage is not available in this mode, and user state must be stored in a Django session, which is much less efficient.

  This is an unusual use case, and should not affect many users.

CKEditor issues

- Scrolling on narrow devices, especially when opening the keyboard inside the CKEditor, does not always work ideally - sometimes the keyboard can appear in the wrong place on-screen.
- Sometimes the CKEditor moves unexpectedly on-screen in use.
- Sometimes in Safari on iOS devices, a rendering bug will apparently truncate or reposition portions of the toolbar when the CKEditor is opened - even though sections may appear to missing or moved, they can still be activated by touching the part of the screen where they should have been found.

Django Admin issues

- In the page tree, the first touch on the page opens the keyboard which may be undesirable. This happens because Django automatically focuses the search form input.

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
```

(continues on next page)

(continued from previous page)

```
            * 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 classes are sub-classes 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 sub-classes 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 %}` template tag, it's called:

- first, by *menus.menu_pool.MenuPool.get_nodes()*, with the argument `post_cut = False`
- later, by the template tag, 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 template tag.

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.

It also has an `attr` attribute, a dictionary that's provided for you to add arbitrary attributes to, rather than placing them directly on the node itself, where they might clash with something.

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. To check if node represents a CMS Page, check for `is_page` in *menus.base.NavigationNode.attr* and that it is `True`.

Menu system logic

Let's look at an example using the {% show_menu %} template tag. It will be different for other template tags, 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 template tag 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 `cms_menus.py`**

        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*); fo

        call its *menus.base.Menu.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()* with `post_cut=Fal`

  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 ==`**

  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 template tag
* ***menus.menu_pool.MenuPool.apply_modifiers()* with `post_cut = True` loops over all the M**

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

    menus.modifiers.Level.mark_levels()

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

Some commonly-used plugins

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

Please note that dozens if not hundreds of different django CMS plugins have been made available under open-source licences. Some, like the ones on this page, are likely to be of general interest, while others are highly specialised.

This page only lists those that fall under the responsibility of the django CMS project. Please see the Django Packages site for some more, or just do a web search for the functionality you seek - you'll be surprised at the range of plugins that has been created.

django CMS Core Addons

We maintain a set of *Core Addons* for django CMS.

You don't need to use them, and for many of them alternatives exist, but they represent a good way to get started with a reliable project set-up. We recommend them for new users of django CMS in particular. For example, if you start a project on Divio Cloud or using the django CMS installer, this is the set of addons you'll have installed by default.

The django CMS Core Addons are:

- Django Filer - a file management application for images and other documents.
- django CMS Admin Style - a CSS theme for the Django admin
- django CMS Text CKEditor - our default rich text WYSIYG editor
- django CMS Link - add links to content
- django CMS Picture - add images to your site (Filer-compatible)
- django CMS File - add files or an entire folder to your pages (Filer-compatible)
- django CMS Style - create HTML containers with classes, styles, ids and other attributes
- django CMS Snippet - insert arbitrary HTML content
- django CMS Audio - publish audio files (Filer-compatible)
- django CMS Video - embed videos from YouTube, Vimeo and other services, or use uploaded videos (Filer-compatible)
- django CMS GoogleMap - displays a map of an address on your page. Supports addresses and co-ordinates. Zoom level and route planner options can also be set.

We welcome feedback, documentation, patches and any other help to maintain and improve these valuable components.

Other addons of note

These packages are no longer officially guaranteed support by the django CMS project, but they have good community support.

- django CMS Inherit - renders the plugins from a specified page (and language) in its place
- django CMS Column - layout page content in columns
- django CMS Teaser - displays a teaser box for another page or a URL, complete with picture and a description

Deprecated addons

Some older plugins that you may have encountered are now deprecated and we advise against incorporating them into new projects.

These are:

- cmsplugin-filer
- Aldryn Style
- Aldryn Locations
- Aldryn Snippet

Search and django CMS

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

4.1.4 Reference

Technical reference material.

API References

cms.api

Python APIs for creating CMS content. 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:

```
def my_function():
    from cms.api import api_function

    api_function(...)
```

instead of:

```
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 visibility to authenticated users.

cms.api.**VISIBILITY_ANONYMOUS**
Used for the `limit_menu_visibility` keyword argument to *create_page()*. Limits menu visibility 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*, *publication_end_date=None*, *in_navigation=False*, *soft_root=False*, *reverse_id=None*, *navigation_extenders=None*, *published=False*, *site=None*, *login_required=False*, *limit_visibility_in_menu=VISIBILITY_ALL*, *position="last-child"*, *overwrite_url=None*, *xframe_options=Page.X_FRAME_OPTIONS_INHERIT*)
Creates a *cms.models.Page* instance and returns it. Also creates a *cms.models.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* sub-class) – 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.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 soft root 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
- **xframe_options** (*int*) – X Frame Option value for Clickjacking protection

cms.api.**create_title**(*language*, *title*, *page*, *menu_title=None*, *slug=None*, *redirect=None*, *meta_description=None*, *parent=None*, *overwrite_url=None*)
Creates a *cms.models.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.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.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* sub-class, 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-treebeard `pos` value for `treebeard.models.Node.add_sibling()`
- **target** – Parent plugin. Must be plugin instance
- **data** – 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.PagePermission* object that gets created.

Parameters

- **page** (*cms.models.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.ACCESS_PAGE*, *cms.models.ACCESS_CHILDREN*, *cms.models.ACCESS_DESCENDANTS* or *cms.models.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.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.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.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:

```
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.constants.**EXPIRE_NOW**
Constant of 0 (zero) used for cache control headers

cms.constants.**MAX_EXPIRATION_TTL**
Constant of 31536000 or 365 days in seconds used for cache control headers

cms.app_base

class cms.app_base.**CMSApp**

_urls
list of urlconfs: example: `_urls = ["myapp.urls"]`

_menus
list of menu classes: example: `_menus = [MyAppMenu]`

name = None
name of the apphook (required)

app_name = None
name of the app, this enables Django namespaces support (optional)

app_config = None
configuration model (optional)

permissions = True
if set to true, apphook inherits permissions from the current page

exclude_permissions = []
list of application names to exclude from inheriting CMS permissions

get_configs()
Returns all the apphook configuration instances.

get_config(*namespace*)
Returns the apphook configuration instance linked to the given namespace

get_config_add_url()
Returns the url to add a new apphook configuration instance (usually the model admin add view)

get_menus(*page*, *language*, ***kwargs*)
New in version 3.3: `CMSApp.get_menus` accepts page, language and generic keyword arguments: you can customize this function to return different list of menu classes according to the given arguments.

Returns the menus for the apphook instance, selected according to the given arguments.

By default it returns the menus assigned to *_menus*

If no page and language are provided, this method **must** return all the menus used by this apphook. Example:

```
if page and page.reverse_id == 'page1':
    return [Menu1]
elif page and page.reverse_id == 'page2':
    return [Menu2]
else:
    return [Menu1, Menu2]
```

param page page the apphook is attached to
param language current site language
return list of menu classes

get_urls (*page*, *language*, ***kwargs*)
New in version 3.3.

Returns the URL configurations for the apphook instance, selected according to the given arguments.

By default it returns the urls assigned to *_urls*

This method **must** return a non empty list of URL configurations, even if no arguments are passed.

Parameters
- **page** – page the apphook is attached to
- **language** – current site language

Returns list of strings representing URL configurations

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 sub-commands:

- `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 therefore plugins), which includes `cms moderator on` as well as page copy operations in the admin.

It is recommended 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 sub-commands:

- `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

The `copy` command is used to copy content from one language or site to another.

It has two sub-commands:

- `cms copy lang` copy content to a given language.
- `cms copy site` copy pages and content to a given site.

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-lang`: the language to copy the content from;
- `--to-lang`: the language to copy the content to.

It accepts the following options

- `--force`: set to copy content even if a placeholder already has content; if set, copied content will be appended to the original one;
- `--site`: specify a SITE_ID to operate on sites different from the current one;
- `--verbosity`: set for more verbose output.
- `--skip-content`: if set, content is not copied, and the command will only create titles in the given language.

Example:

```
cms copy lang --from-lang=en --to-lang=de --force --site=2 --verbosity=2
```

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 placeholders 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 --from-site=1 --to-site=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.
- `-l`, `--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
cms 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-tree

Occasionally, the pages and plugins tree can become corrupted. Typical symptoms include problems when trying to copy or delete plugins or pages.

This command will fix small corruptions by rebuilding the tree.

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.

This command has been removed in *django CMS 3.1* and replaced with *fix-tree*.

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`.

The `MIDDLEWARE_CLASSES` setting

`cms.middleware.utils.ApphookReloadMiddleware`

Adding `ApphookReloadMiddleware` to the `MIDDLEWARE_CLASSES` tuple will enable automatic server restarts when changes are made to apphook configurations. It should be placed as near to the top of the classes as possible.

Note: This has been tested and works in many production environments and deployment configurations, but we haven't been able to test it with all possible set-ups. Please file an issue if you discover one where it fails.

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.

django CMS 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_full_name()` method.

The models must also be editable via Django's 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 behaviour 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 an example, see *Templates*.

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 customised 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:::

```
# -*- 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 = {
    None: {
        "plugins": ['TextPlugin'],
        'excluded_plugins': ['InheritPlugin'],
    },
    '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,
        },
```

(continues on next page)

(continued from previous page)

```
        '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 define plugins in a granular fashion, as shown above with `base.html content`.

Configuration is retrieved in the following order:

- CMS_PLACEHOLDER_CONF['template placeholder']
- CMS_PLACEHOLDER_CONF['placeholder']
- CMS_PLACEHOLDER_CONF['template']
- CMS_PLACEHOLDER_CONF[None]

The first `CMS_PLACEHOLDER_CONF` key that matches for the required configuration attribute is used.

E.g: given the example above if the `plugins` configuration is retrieved for the `content` placeholder in a page using the `base.html` template, the value `['TextPlugin', 'PicturePlugin', 'TeaserPlugin']` will be returned as `'base.html content'` matches; if the same configuration is retrieved for the `content` placeholder in a page using `fullwidth.html` template, the returned value will be `['TextPlugin', 'PicturePlugin']`. If `plugins` configuration is retrieved for `sidebar_left` placeholder, `['TextPlugin']` from `CMS_PLACEHOLDER_CONF` key `None` will be returned.

`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.

`excluded_plugins` A list of plugins that will not be added to the given placeholder; this takes precedence over `plugins` configuration: if a plugin is present in both lists, it **will not** be available in the placeholder. This is basically a way to **blacklist** a plugin: even if registered, it will not be available in the placeholder. If set on the `None` (default) key, the plugins will not be available in any placeholder (except the `excluded_plugins` configuration is overridden in more specific `CMS_PLACEHOLDER_KEYS`.

`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 internationalised.

`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 automatically 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 text plugin: `{'body':'<p>Lorem ipsum</p>'}` Example for a link plugin: `{'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:

```
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 overwrites the `plugins` setting to allow `TeaserPlugin`, thus you have not to duplicate the configuration of `content`.

CMS_PLUGIN_CONTEXT_PROCESSORS

default `[]`

A list of plugin context processors. Plugin context processors are callables that modify all plugins' context *before* rendering. See *How to create 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 *How to create custom Plugins* for more information.

CMS_APPHOOKS

default: `()`

A list of import paths for *`cms.app_base.CMSApp`* sub-classes.

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',
)
```

Internationalisation and localisation (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,
```

(continues on next page)

(continued from previous page)

```
            '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 behaviour 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 behaviour.

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

Hides untranslated pages in menus.

When applied to the `default` directive, if `False`, all pages in menus will be listed in all languages, including those that don't yet have content in a particular language. If `True`, untranslated pages will be hidden.

When applied to a particular language, hides that language's pages in menus until translations exist for them.

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

If your site has languages which use non-ASCII character sets, *CMS_UNIHANDECODE_HOST* and *CMS_UNIHANDECODE_VERSION* will allow it to automate slug generation for those languages too.

Support for this is provided by the unihandecode.js project.

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_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 sub-folders numbered with the id of the page.

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

Advanced Settings

CMS_INTERNAL_IPS

default `[]`

By default `CMS_INTERNAL_IPS` is an empty list (`[]`).

If left as an empty list, this setting does not add any restrictions to the toolbar. However, if set, the toolbar will only appear for client IP addresses that are in this list.

This setting may also be set to an *IpRangeList* from the external package `iptools`. This package allows convenient syntax for defining complex IP address ranges.

The client IP address is obtained via the *CMS_REQUEST_IP_RESOLVER* in the `cms.middleware.toolbar.ToolbarMiddleware` middleware.

CMS_REQUEST_IP_RESOLVER

default 'cms.utils.request_ip_resolvers.default_request_ip_resolver'

This setting is used system-wide to provide a consistent and plug-able means of extracting a client IP address from the HTTP request. The default implementation should work for most project architectures, but if not, the administrator can provide their own method to handle the project's specific circumstances.

The supplied method should accept a single argument *request* and return an IP address String.

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.

Changed in version 3.2.1:: CMS_RAW_ID_USERS also applies to `GlobalPagePermission` admin.

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 time zone 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 template tags be cached? Takes the current language and time zone 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_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 third-party apps will be loaded.

Example:

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

(continues on next page)

(continued from previous page)

```
    'cms.cms_toolbars.PageToolbar',

    # third-party Toolbar
    'aldryn_blog.cms_toolbars.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`

By default, the django CMS toolbar is displayed to logged-in admin users on all pages that use the `{% cms_toolbar %}` template tag. Its appearance can be optionally restricted to django CMS pages only (technically, pages that are rendered by a django CMS view).

When this is set to `True`, all other pages will no longer display the toolbar. This includes pages with apphooks applied to them, as they are handled by the other application's views, and not django CMS's.

Changed in version 3.2.1:: CMS_APP_NAME has been removed as it's no longer required.

CMS_DEFAULT_X_FRAME_OPTIONS

default `constants.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 `cms.constants` e.g.

- X_FRAME_OPTIONS_INHERIT
- X_FRAME_OPTIONS_ALLOW
- X_FRAME_OPTIONS_SAMEORIGIN
- X_FRAME_OPTIONS_DENY

CMS_TOOLBAR_SIMPLE_STRUCTURE_MODE

default: `True`

The new structure board operates by default in "simple" mode. The older mode used absolute positioning. Setting this attribute to `False` will allow the absolute positioning used in versions prior to 3.2. This setting will be removed in 3.3.

Example:

```
CMS_TOOLBAR_SIMPLE_STRUCTURE_MODE = False
```

CMS_PAGE_WIZARD_DEFAULT_TEMPLATE

default `TEMPLATE_INHERITANCE_MAGIC`

This is the path of the template used to create pages in the wizard. It must be one of the templates in *CMS_TEMPLATES*.

CMS_PAGE_WIZARD_CONTENT_PLACEHOLDER

default None

When set to an editable, non-static placeholder that is available on the page template, the CMS page wizards will target the specified placeholder when adding any content supplied in the wizards' "Content" field. If this is left unset, then the content will target the first suitable placeholder found on the page's template.

CMS_PAGE_WIZARD_CONTENT_PLUGIN

default `TextPlugin`

This is the name of the plugin created in the Page Wizard when the "Content" field is filled in. There should be no need to change it, unless you **don't** use `djangocms-text-ckeditor` in your project.

CMS_PAGE_WIZARD_CONTENT_PLUGIN_BODY

default `body`

This is the name of the body field in the plugin created in the Page Wizard when the "Content" field is filled in. There should be no need to change it, unless you **don't** use `djangocms-text-ckeditor` in your project **and** your custom plugin defined in *CMS_PAGE_WIZARD_CONTENT_PLUGIN* have a body field **different** than `body`.

Form and model fields

Model fields

class cms.models.fields.**PageField**
: This is a foreign key field to the *cms.models.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.ForeignKey` but does not require the `othermodel` argument.

class cms.models.fields.**PlaceholderField**
: A foreign key field to the *cms.models.placeholdermodel.Placeholder* model.

Form fields

class cms.forms.fields.**PageSelectFormField**
: Behaves like a `django.forms.ModelChoiceField` field for the *cms.models.Page* model, but displays itself as a split field with a select drop-down 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 drop-down is easier to use. This takes the same arguments as `django.forms.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.

Menus and 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 template tags, you need to have `{% load menu_tags %}` in your template before the line on which you call the template tag.

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 six optional parameters: `start_level`, `end_level`, `extra_inactive`, `extra_active`, `namespace` and `root_id`.

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.

The fifth parameter, `namespace`, is currently not implemented.

The sixth parameter `root_id` specifies the id of the root node.

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

Some Examples

Complete navigation (as a nested list):

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

Navigation with active tree (as a nested list):

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

Navigation with only one active extra level:

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

Level 1 navigation (as a nested list):

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

Navigation with a custom template:

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

show_menu_below_id

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

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

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

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

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:

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

Rooted at level 0:

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

Or with a custom template:

```
<ul>
    {% show_sub_menu 1 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 *How to customise navigation menus* documentation

Menu system classes and function

`menu` application

class menus.base.**Menu**
The base class for all menu-generating classes.

get_nodes(*self*, *request*)
Each sub-class of `Menu` should return a list of NavigationNode instances.

class menus.base.**Modifier**
The base class for all menu-modifying classes. A modifier add, removes or changes NavigationNodes in the list.

modify(*self*, *request*, *nodes*, *namespace*, *root_id*, *post_cut*, *breadcrumb*)
Each sub-class of `Modifier` should implement a `modify()` method.

class menus.menu_pool.**MenuPool**

get_nodes()

discover_menus()

apply_modifiers()

_build_nodes()

_mark_selected()

menus.menu_pool.**_build_nodes_inner_for_one_menu**()

menus.templatetags.menu_tags.**cut_levels**()

class menus.templatetags.menu_tags.**ShowMenu**

get_context()

class menus.base.**NavigationNode**(*title, url, id[, parent_id=None][, parent_namespace=None][, attr=None][, visible=True]*)
Each node in a menu tree is represented by a `NavigationNode` instance.

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.

attr
A dictionary, provided in order that arbitrary attributes may be added to the node - placing them directly on the node itself could cause a clash with an existing or future attribute.

An important key in this dictionary is `is_page`: if `True`, the node represents a django CMS `Page` object.

Nodes that represent CMS pages have the following keys in `attr`:

- **auth_required** (*bool*) – is authentication required to access this page
- **is_page** (*bool*) – Always True
- **navigation_extenders** (*list*) – navigation extenders connected to this node
- **redirect_url** (*str*) – redirect URL of page (if any)
- **reverse_id** (*str*) – unique identifier for the page
- **soft_root** (*bool*) – whether page is a soft root
- **visible_for_authenticated** (*bool*) – visible for authenticated users
- **visible_for_anonymous** (*bool*) – visible for anonymous users

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.Page*.

attr
A dictionary, provided in order that arbitrary attributes may be added to the node - placing them directly on the node itself could cause a clash with an existing or future attribute.

An important key in this dictionary is `is_page`: if `True`, the node represents a django CMS `Page` object.

class menus.modifiers.**Marker**

class menus.modifiers.**AuthVisibility**

class menus.modifiers.**Level**

mark_levels()

cms application

class cms.menu.**CMSMenu**
Subclass of *menus.base.Menu*. Its *get_nodes()* creates a list of NavigationNodes based on `Page` objects.

class cms.menu.**NavExtender**

class cms.menu.**SoftRootCutter**

class cms.menu_bases.**CMSAttachMenu**

Models

class cms.models.**Page**
A `Page` is the basic unit of site structure in django CMS. The CMS uses a hierarchical page model: each page stands in relation to other pages as parent, child or sibling. This hierarchy is managed by the django-treebeard library.

A `Page` also has language-specific properties - for example, it will have a title and a slug for each language it exists in. These properties are managed by the *cms.models.Title* model.

Permissions

class cms.models.**PagePermission**

cms.models.**ACCESS_PAGE**

cms.models.**ACCESS_CHILDREN**

cms.models.**ACCESS_DESCENDANTS**

cms.models.**ACCESS_PAGE_AND_DESCENDANTS**

Placeholders

class cms.models.placeholdermodel.**Placeholder**
Placeholders can be filled with plugins, which store or generate content.

class cms.admin.placeholderadmin.**PlaceholderAdminMixin**

Plugins

CMSPluginBase Attributes and Methods Reference

class cms.plugin_base.**CMSPluginBase**
Inherits django.contrib.admin.ModelAdmin and in most respects behaves like a normal sub-class. Note however that some attributes of ModelAdmin simply won't make sense in the context of a Plugin.

Attributes

admin_preview
Default: False

If True, displays a preview in the admin.

allow_children
Default: False

Allows this plugin to have child plugins - other plugins placed inside it?

If True you need to ensure that your plugin can render its children in the plugin template. For example:

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

instance.child_plugin_instances provides access to all the plugin's children. They are pre-filled and ready to use. The child plugins should be rendered using the {% render_plugin %} template tag.

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.

If present and set to False, the plugin will prevent the caching of the resulting page.

Important: Setting this to False will effectively disable the CMS page cache and all upstream caches for pages where the plugin appears. This may be useful in certain cases but for general cache management, consider using the much more capable *get_cache_expiration()*.

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.

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

frontend_edit_template
This attribute is deprecated and will be removed in 3.5.

Default: `cms/toolbar/plugin.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*.

module
Will group the plugin in the plugin picker. If module is `None`, plugin is listed in the "Generic" group.

name
Will be displayed in the plugin picker.

page_only
Default: `False`

Set to `True` if this plugin should only be used in a placeholder that is attached to a django CMS page, and not other models with `PlaceholderFields`.

See also: *child_classes*, *parent_classes*, *require_parent*.

parent_classes
Default: `None`

A list of the names of permissible parent classes for this plugin.

See also: *child_classes*, *require_parent*.

render_plugin
If set to `False`, this plugin will not be rendered at all. Default: `True`

If `True`, *render_template()* must also be defined.

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`

Not all plugins are usable in text plugins. If your plugin *is* usable in a text plugin:
- set this to `True`
- make sure your plugin provides its own *icon_src()*

See also: *icon_src*, *icon_alt*.

Methods

get_plugin_urls(*instance*)
Returns the URL patterns the plugin wants to register views for. They are included under django CMS's page admin URLS in the plugin path (e.g.: `/admin/cms/page/plugin/<plugin-name>/` in the default case).

`get_plugin_urls()` is useful if your plugin needs to talk asynchronously to the admin.

get_render_template()
If you need to determine the plugin render model at render time you can implement the *get_render_template()* method on the plugin class; this method takes 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()*

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(*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()
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.

get_cache_expiration(*self*, *request*, *instance*, *placeholder*)
Provides expiration value to the placeholder, and in turn to the page for determining the appropriate Cache-Control headers to add to the HTTPResponse object.

Must return one of:

None This means the placeholder and the page will not even consider this plugin when calculating the page expiration.

datetime A specific date and time (timezone-aware) in the future when this plugin's content expires.

Important: The returned `datetime` must be timezone-aware or the plugin will be ignored (with a warning) during expiration calculations.

---

int An number of seconds that this plugin's content can be cached.

There are constants are defined in `cms.constants` that may be useful: *EXPIRE_NOW* and *MAX_EXPIRATION_TTL*.

An integer value of `0` (zero) or *EXPIRE_NOW* effectively means "do not cache". Negative values will be treated as *EXPIRE_NOW*. Values exceeding the value *MAX_EXPIRATION_TTL* will be set to that value.

Negative `timedelta` values or those greater than *MAX_EXPIRATION_TTL* will also be ranged in the same manner.

Similarly, `datetime` values earlier than now will be treated as *EXPIRE_NOW*. Values greater than *MAX_EXPIRATION_TTL* seconds in the future will be treated as *MAX_EXPIRATION_TTL* seconds in the future.

Parameters

- **request** – Relevant `HTTPRequest` instance.
- **instance** – The `CMSPlugin` instance that is being rendered.

Return type `None` or `datetime` or `int`

get_vary_cache_on(*self*, *request*, *instance*, *placeholder*)
Returns an HTTP VARY header string or a list of them to be considered by the placeholder and in turn by the page to caching behaviour.

Overriding this method is optional.

Must return one of:

None This means that this plugin declares no headers for the cache to be varied upon. (default)

string The name of a header to vary caching upon.

list of strings A list of strings, each corresponding to a header to vary the cache upon.

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*.

icon_src(*instance*)

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()*

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

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

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:

```
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

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*.

class cms.plugin_base.**PluginMenuItem**

__init__(*name*, *url*, *data*, *question=None*, *action='ajax'*, *attributes=None*)

Creates an item in the plugin / placeholder menu

Parameters

- **name** – Item name (label)
- **url** – URL the item points to. This URL will be called using POST
- **data** – Data to be POSTed to the above URL
- **question** – Confirmation text to be shown to the user prior to call the given URL (optional)
- **action** – Custom action to be called on click; currently supported: 'ajax', 'ajax_add'
- **attributes** – Dictionary whose content will be added as data-attributes to the menu item

CMSPlugin Attributes and Methods Reference

class cms.models.pluginmodel.**CMSPlugin**

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:

```
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.
- *`get_translatable_content()`*

Example:

```
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()`*.

`get_add_url``()`

Returns the URL to call to add a plugin instance; useful to implement plugin-specific logic in a custom view.

`get_edit_url``()`

Returns the URL to call to edit a plugin instance; useful to implement plugin-specific logic in a custom view.

`get_move_url``()`

Returns the URL to call to move a plugin instance; useful to implement plugin-specific logic in a custom view.

`get_delete_url``()`

Returns the URL to call to delete a plugin instance; useful to implement plugin-specific logic in a custom view.

get_copy_url()
Returns the URL to call to copy a plugin instance; useful to implement plugin-specific logic in a custom view.

class cms.plugin_pool.**PluginPool**

Sitemaps

class cms.sitemaps.**CMSSitemap**

Template Tags

CMS template tags

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

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

Placeholders

placeholder

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

The `placeholder` template tag 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 re-size 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 behaviour.

Important: `{% placeholder %}` will only work inside the template's `<body>`.

---

static_placeholder

New in version 3.0.

The `{% static_placeholder %}` template tag can be used anywhere in a template element after the `{% cms_toolbar %}` tag. A static placeholder instance is not bound to any particular page or model - in other words, everywhere it appears, a static placeholder will hold exactly the same content.

The `{% static_placeholder %}` tag is normally used to display the same content on multiple locations or inside of apphooks or other third party apps.

Otherwise, a static placeholder behaves like a "normal" placeholder, to which plugins can be added.

A static placeholder needs to be published to show up on live pages, and requires a name.

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 template tag 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.

Important: `{% static_placeholder %}` will only work inside the template's `<body>`.

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)
- `as` keyword followed by `varname` (optional): the template tag 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 (`en`) 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 template tag 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 template tags 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 template tags:

```
{% 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.

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_full_path %}<link rel="canonical" href="{% page_url request.current_
↪page %}">{% endif %}
```

page_attribute

This template tag 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 template tag 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 template tag acts like the template tag `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 change form 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 navigation block, then the actual plugin contents inside a `DIV.contentgroup-items` block. In this example, the navigation 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 plugin's content, by double-clicking its navigation 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 %} active{% endif %}">
      {% render_plugin child  %}
    </div>{% endfor %}
  </div>
</div>
{% endblock %}
```

New in version 3.0.

render_model

`render_model` is the way to add frontend editing to any Django model. It both renders 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 template tag you can show and edit page titles as well as fields in standard django models, see *How to enable 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><cms-plugin class="cms-plugin cms-plugin-myapp-mymodel-title-1">{{ my_model.title
↪}}</cms-plugin></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 template tag 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` template tag;
- `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 template tag output can be saved as a context variable for later use.

Note: By default this template tag escapes the content of the rendered model attribute. This helps prevent a range of security vulnerabilities stemming from HTML, JavaScript, and CSS Code Injection.

To change this behavior, the project administrator should carefully review each use of this template tag and 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 is satisfied that the content is clean, he or she can add the "safe" filter parameter to the template tag if the content should be rendered without escaping.

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 -->
<template class="cms-plugin cms-plugin-start cms-plugin-myapp-mymodel-1"></template>
    <h1>{{ my_model.title }}</h1>
    <div class="body">
        {{ my_model.date|date:"d F Y" }}
        {{ my_model.text }}
    </div>
<template class="cms-plugin cms-plugin-end cms-plugin-myapp-mymodel-1"></template>
```

In the block the `my_model` is aliased as `instance` and every attribute and method is available; also template tags and filters are available in the block.

Warning: If the `{% render_model_block %}` contains template tags or template code that rely on or manipulate context data that the `{% render_model_block %}` also makes use of, you may experience some unexpected effects. Unless you are sure that such conflicts will not occur it is advised to keep the code within a `{% render_model_block %}` as simple and short as possible.

Arguments:

- `instance`: instance of your model in the template
- `edit_fields` (optional): a comma separated list of fields editable in the popup editor; when template tag 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 template tag output can be saved as a context variable for later use.

Note: By default this template tag escapes the content of the rendered model attribute. This helps prevent a range of security vulnerabilities stemming from HTML, JavaScript, and CSS Code Injection.

To change this behavior, the project administrator should carefully review each use of this template tag and 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 is satisfied that the content is clean, he or she can add the "safe" filter parameter to the template tag if the content should be rendered without escaping.

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 change form for the provided fields.

```
<h3><a href="{{ my_model.get_absolute_url }}">{{ my_model.title }}</a> {% render_
↪model_icon my_model %}</h3>
```

It will render to something like:

```
<h3>
    <a href="{{ my_model.get_absolute_url }}">{{ my_model.title }}</a>
    <template class="cms-plugin cms-plugin-start cms-plugin-myapp-mymodel-1 cms-
↪render-model-icon"></template>
        <!-- The image below is the active area that triggers the frontend editor -->
        <img src="/static/cms/img/toolbar/render_model_placeholder.png">
    <template class="cms-plugin cms-plugin-end cms-plugin-myapp-mymodel-1 cms-render-
↪model-icon"></template>
</h3>
```

Note: Icon and position can be customised 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 template tag 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 template tag output can be saved as a context variable for later use.

Note: By default this template tag escapes the content of the rendered model attribute. This helps prevent a range of security vulnerabilities stemming from HTML, JavaScript, and CSS Code Injection.

To change this behavior, the project administrator should carefully review each use of this template tag and 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 is satisfied that the content is clean, he or she can add the "safe" filter parameter to the template tag if the content should be rendered without escaping.

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 add form for the provided model.

```
<h3><a href="{{ my_model.get_absolute_url }}">{{ my_model.title }}</a> {% render_
↪model_add my_model %}</h3>
```

It will render to something like:

```
<h3>
    <a href="{{ my_model.get_absolute_url }}">{{ my_model.title }}</a>
    <template class="cms-plugin cms-plugin-start cms-plugin-myapp-mymodel-1 cms-
↪render-model-add"></template>
        <!-- The image below is the active area that triggers the frontend editor -->
        <img src="/static/cms/img/toolbar/render_model_placeholder.png">
    <template class="cms-plugin cms-plugin-end cms-plugin-myapp-mymodel-1 cms-render-
↪model-add"></template>
</h3>
```

Note: Icon and position can be customised 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 template tag output can be saved as a context variable for later use.

Note: By default this template tag escapes the content of the rendered model attribute. This helps prevent a range of security vulnerabilities stemming from HTML, JavaScript, and CSS Code Injection.

To change this behavior, the project administrator should carefully review each use of this template tag and 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 is satisfied that the content is clean, he or she can add the "safe" filter parameter to the template tag if the content should be rendered without escaping.

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 template tag 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:

```
<template class="cms-plugin cms-plugin-start cms-plugin-myapp-mymodel-1 cms-render-
↪model-add"></template>
    <div>New Object</div>
<template class="cms-plugin cms-plugin-end cms-plugin-myapp-mymodel-1 cms-render-
↪model-add"></template>
```

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 template tag 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 template tag 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 `menus.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 template tags

The `cms_toolbar` template tag is included in the `cms_tags` library and will add the required CSS and javascript to the sekizai blocks in the base template. The template tag must be placed before any `{% placeholder %}` occurrences within your HTML.

Important: `{% cms_toolbar %}` will only work correctly inside the template's `<body>`.

Example:

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

Note: Be aware that you cannot 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.

Titles

class cms.models.**Title**

Titles support pages by providing a storage mechanism, amongst other things, for language-specific properties of *cms.models.Page*, such as its title, slug, meta description and so on.

Each `Title` has a foreign key to *cms.models.Page*; each *cms.models.Page* may have several `Titles`.

The 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 models this toolbar works on; used for redirection 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 built-in 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.

get_menu(*self*, *key*, *verbose_name=None*, *side=LEFT*, *position=None*)
If a menu with `key` already exists, this method will return that menu.

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

Important: Overlay and **sideframe**

Then django CMS *sideframe* has been replaced with an *overlay* mechanism. The API still refers to the `sideframe`, because it is invoked in the same way, and what has changed is merely the behaviour in the user's browser.

In other words, *sideframe* and the *overlay* refer to different versions of the same thing.

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.

get_alphabetical_insert_position(*self*, *new_menu_name*, *item_type*, *default=0*)

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 sub-class 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 sideframe and returns it.

`on_close` can be set to `None` to do nothing when the sideframe 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 sideframe.

`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 sub-classes 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 sideframe.

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 selected.

class cms.toolbar_pool.**ToolbarPool**

register(*self*, *toolbar*)
Register this toolbar.

class cms.extensions.toolbar.**ExtensionToolbar**

get_page_extension_admin()

_setup_extension_toolbar()

get_title_extension_admin()

New in version 3.2.

Content creation wizards

See the *How-to section on wizards* for an introduction to creating wizards.

Wizard classes are sub-classes of `cms.wizards.wizard_base.Wizard`.

They need to be registered with the `cms.wizards.wizard_pool.wizard_pool`:

```
wizard_pool.register(my_app_wizard)
```

Finally, a wizard needs to be instantiated, for example:

```
my_app_wizard = MyAppWizard(
    title="New MyApp",
    weight=200,
    form=MyAppWizardForm,
    description="Create a new MyApp instance",
)
```

When instantiating a Wizard object, use the keywords:

title The title of the wizard. This will appear in a large font size on the wizard "menu"

weight The "weight" of the wizard when determining the sort-order.

form The form to use for this wizard. This is mandatory, but can be sub-classed from *django.forms.form* or *django.forms.ModelForm*.

model If a Form is used above, this keyword argument must be supplied and should contain the model class. This is used to determine the unique wizard "signature" amongst other things.

template_name An optional template can be supplied.

description The description is optional, but if it is not supplied, the CMS will create one from the pattern: "Create a new «model.verbose_name» instance."

edit_mode_on_success If set, the CMS will switch the user to edit-mode by adding an `edit` param to the query-string of the URL returned by `get_success_url`. This is `True` by default.

Base Wizard

All wizard classes should inherit from `cms.wizards.wizard_base.Wizard`. This class implements a number of methods that may be overridden as required.

Base Wizard methods

get_description

Simply returns the `description` property assigned during instantiation or one derived from the model if description is not provided during instantiation. Override this method if this needs to be determined programmatically.

get_title

Simply returns the `title` property assigned during instantiation. Override this method if this needs to be determined programmatically.

get_success_url

Once the wizard has completed, the user will be redirected to the URL of the new object that was created. By default, this is done by return the result of calling the `get_absolute_url` method on the object. This may then be modified to force the user into edit mode if the wizard property `edit_mode_on_success` is True.

In some cases, the created content will not implement `get_absolute_url` or that redirecting the user is undesirable. In these cases, simply override this method. If `get_success_url` returns `None`, the CMS will just redirect to the current page after the object is created.

This method is called by the CMS with the parameter:

obj The created object

kwargs Arbitrary keyword arguments

get_weight

Simply returns the `weight` property assigned during instantiation. Override this method if this needs to be determined programmatically.

user_has_add_permission

This should return a boolean reflecting whether the user has permission to create the underlying content for the wizard.

This method is called by the CMS with these parameters:

user The current user

page The current CMS page the user is viewing when invoking the wizard

wizard_pool

`wizard_pool` includes a read-only property `discovered` which returns the Boolean `True` if wizard-discovery has already occurred and `False` otherwise.

Wizard pool methods

is_registered

Sometimes, it may be necessary to check to see if a specific wizard has been registered. To do this, simply call:

```
value = wizard_pool.is_registered(«wizard»)
```

register

You may notice from the example above that the last line in the sample code is:

```
wizard_pool.register(my_app_wizard)
```

This sort of thing should look very familiar, as a similar approach is used for cms_apps, template tags and even Django's admin.

Calling the wizard pool's `register` method will register the provided wizard into the pool, unless there is already a wizard of the same module and class name. In this case, the register method will raise a `cms.wizards.wizard_pool.AlreadyRegisteredException`.

unregister

It may be useful to unregister wizards that have already been registered with the pool. To do this, simply call:

```
value = wizard_pool.unregister(«wizard»)
```

The value returned will be a Boolean: `True` if a wizard was successfully unregistered or `False` otherwise.

get_entry

If you would like to get a reference to a specific wizard in the pool, just call `get_entry()` as follows:

```
wizard = wizard_pool.get_entry(my_app_wizard)
```

get_entries

`get_entries()` is useful if it is required to have a list of all registered wizards. Typically, this is used to iterate over them all. Note that they will be returned in the order of their `weight`: smallest numbers for weight are returned first.:

```
for wizard in wizard_pool.get_entries():
    # do something with a wizard...
```

4.1.5 Development & community

django CMS is an open-source project, and relies on its community of users to keep getting better.

The contributors to django CMS come from across the world, and have 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 discussions, filing bug reports, proposing improvements, contributing code or documentation, and testing the

django CMS's development community

You can join us online:

- in our IRC channel, #django-cms, on `irc.freenode.net`. If you don't have an IRC client, you can join our IRC channel using the KiwiIRC web client, which works pretty well.
- 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 can also follow:

- the Travis Continuous Integration build reports
- the @djangocms Twitter account for general announcements

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*.

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.

django CMS represents Divio's vision for a general-purpose CMS platform able to meet its needs as a web agency with a large portfolio of different clients. This vision continues to guide the development of django CMS.

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. As the chief backer of django CMS, and in order to ensure a consistent and long-term approach to the project, Divio reserves the right of final say in any decisions concerning its development.

Divio remains thoroughly committed to django CMS both as a high-quality technical product and as a healthy open source project.

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.

Current core developers

- Angelo Dini https://github.com/finalangel
- Daniele Procida https://github.com/evildmp
- Iacopo Spalletti https://github.com/yakky
- Jonas Obrist https://github.com/ojii
- Martin Koistinen https://github.com/mkoistinen
- Patrick Lauber https://github.com/digi604
- Paulo Alvarado https://github.com/czpython
- Stefan Foulis https://github.com/stefanfoulis
- Vadim Sikora https://github.com/vxsx

Core designers

django CMS also receives important contributions from *core designers*, responsible for key aspects of its visual design:

- Christian Bertschy
- Matthias Nüesch

Retired core developers

- Chris Glass https://github.com/chrisglass
- Øyvind Saltvik https://github.com/fivethreeo
- Benjamin Wohlwend https://github.com/piquadrat

Following a year or so of inactivity, a core developer will be moved to the "Retired" list, with the understanding that they can rejoin the project in the future whenever they choose.

Becoming a core developer

Anyone can become a core developer. You don't need to be an expert developer, or know more than anyone else about the internals of the CMS. You just have to be a regular contributor, who makes a sustained and valuable contribution to the project.

This contribution can take many forms - not just commits to our codebase. For example, documentation is a valuable contribution, and so is simply being a member of the community who spends time assisting others.

Any member of the core team can nominate a new person for membership. The nomination will be discussed by the technical board, and assuming there are no objections raised, approved.

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 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
- Christian Bertschy
- Daniele Procida (Chair)
- Iacopo Spalletti
- Jonas Obrist
- Martin Koistinen
- Matteo Larghi

The board will co-opt new members as appropriate.

Development policies

Reporting security issues

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.

Review

All patches should be made as pull requests to the GitHub repository. Patches should never be pushed directly.

Nothing may enter the code-base, *including the documentation*, without proper review and formal approval from the core team.

Reviews are welcomed by all members of the community. You don't need to be a core developer, or even an experienced programmer, to contribute usefully to code review. Even noting that you don't understand something in a pull request is valuable feedback and will be taken seriously.

Formal approval

Formal approval means "OK to merge" comments, following review, from at least two different members of the core team who have expertise in the relevant areas, and excluding the author of the pull request.

The exceptions to this are frontend code and documentation, where one "OK to merge" comment will suffice, at least until the team has more expert developers in those areas.

Proposal and discussion of significant changes

New features and backward-incompatible changes should be proposed using the django CMS developers email list. Discussion should take place there before any pull requests are made.

This is in the interests of openness and transparency, and to give the community a chance to participate in and understand the decisions taken by the project.

Release schedule

The roadmap can be found on our website.

We are planning releases according to **key principles and aims**. Issues within milestones are therefore subject to change.

The django CMS developers email list serves as gathering point for developers. We submit ideas and proposals prior to the roadmap goals.

django CMS 3.4 will be the first "LTS" ("Long-Term Support") release of the application. *Long-term support* means that this version will continue to receive security and other critical updates for 24 months after its first release.

Any updates it does receive will be backward-compatible and will not alter functional behaviour. This means that users can deploy this version confident that keeping it up-to-date requires only easily-applied security and other critical updates, until the next LTS release.

Branches

Changed in version 3.3: Previously, we maintained a `master` branch (now deleted), and a set of `support` branches (now pruned, and renamed `release`).

We maintain a number of branches on our GitHub repository.

the latest (highest-numbered) `release/x.y.z` This is the branch that will become the next release on PyPI.

Fixes and backwards-compatible improvements (i.e. most pull requests) will be made against this branch.

`develop` This is the branch that will become the next release that increments the `x` or `y` of the latest `release/x.y.z`.

This branch is for **new features and backwards-incompatible changes**. By their nature, these will require more substantial team co-ordination.

Older `release/x.y.z` branches These represent the final point of development (the highest `y` of older versions). Releases in the full set of older versions have been tagged (use Git Tags to retrieve them).

These branches will only rarely be patched, with security fixes representing the main reason for a patch.

Commits in `release/x.y.z` will be merged forward into `develop` periodically by the core developers.

If in doubt about which branch to work from, ask on the #django-cms IRC channel on freenode or the django-cms-developers email list!

Commits

New in version 3.3.

Commit messages

Commit messages and their subject lines should be written in the past tense, not present tense, for example:

Updated contribution policies.

- Updated branch policy to clarify purpose of develop/release branches
- Added commit policy.
- Added changelog policy.

Keep lines short, and within 72 characters as far as possible.

Squashing commits

In order to make our Git history more useful, and to make life easier for the core developers, please rebase and squash your commit history into a single commit representing a single coherent piece of work.

For example, we don't really need or want a commit history, for what ought to be a single commit, that looks like (newest last):

```
2dceb83 Updated contribution policies.
ffe5f2c Fixed spelling mistake in contribution policies.
29168da Fixed typo.
85d925c Updated commit policy based on feedback.
```

The bottom three commits are just noise. They don't represent development of the code base. The four commits should be squashed into a single, meaningful, commit:

```
85d925c Updated contribution policies.
```

How to squash commits

In this example above, you'd use `git rebase -i HEAD~4` (the `4` refers to the number of commits being squashed - adjust it as required).

This will open a `git-rebase-todo` file (showing commits with the newest last):

```
pick 2dceb83 Updated contribution policies.
pick ffe5f2c Fixed spelling mistake in contribution policies.
pick 29168da Fixed typo.
pick 85d925c Updated commit policy based on feedback.
```

"Fixup" the last three commits, using `f` so that they are squashed into the first, and their commit messages discarded:

```
pick 2dceb83 Updated contribution policies.
f ffe5f2c Fixed spelling mistake in contribution policies.
f 29168da Fixed typo.
f 85d925c Updated commit policy based on feedback.
```

Save - and this will leave you with a single commit containing all of the changes:

```
85d925c Updated contribution policies.
```

Ask for help if you run into trouble!

Changelog

New in version 3.3.

Every new feature, bugfix or other change of substance must be represented in the CHANGELOG. This includes documentation, but **doesn't** extend to things like reformatting code, tidying-up, correcting typos and so on.

Each line in the changelog should begin with a verb in the past tense, for example:

```
* Added CMS_WIZARD_CONTENT_PLACEHOLDER setting
* Renamed the CMS_WIZARD_* settings to CMS_PAGE_WIZARD_*
* Deprecated the old-style wizard-related settings
* Improved handling of uninstalled apphooks
* Fixed an issue which could lead to an apphook without a slug
* Updated contribution policies documentation
```

New lines should be added to the top of the list.

Contributing code

Like every open-source project, django CMS is always looking for motivated individuals to contribute to its source code.

In a nutshell

Here's what the contribution process looks like in brief:

1. Fork our GitHub repository, https://github.com/divio/django-cms
2. Work locally and push your changes to your repository.

3. When you feel your code is good enough for inclusion, send us a pull request.

See the *How to contribute a patch* how-to document for a walk-through of this process.

Basic requirements and standards

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 too inflexible about it.

HTML, CSS and JavaScript

As of django CMS 3.2, we are using the same guidelines as described in Aldryn Boilerplate

Frontend code should be formatted for readability. If in doubt, follow existing examples, or ask.

JS Linting

JavaScript is linted using ESLint. In order to run the linter you need to do this:

```
gulp lint
```

Or you can also run the watcher by just running `gulp`.

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.

If you don't have an IRC client, you can join our IRC channel using the KiwiIRC web client, which works pretty well.

Frontend

Important: When we refer to the *frontend* here, we **only** mean the frontend of django CMS's admin/editor interface.

The frontend of a django CMS website, as seen by its visitors (i.e. the published site), is *wholly independent of this*. django CMS places almost no restrictions at all on the frontend - if a site can be described in HTML/CSS/JavaScript, it can be developed in django CMS.

In order to be able to work with the frontend tooling contributing to the django CMS you need to have the following dependencies installed:

1. Node version 0.12.7 (will install npm as well). We recommend using NVM to get the correct version of Node.
2. gulp - see Gulp's Getting Started notes
3. Local dependencies `npm install`

Styles

We use Sass for our styles. The files are located within `cms/static/cms/sass` and can be compiled using the libsass implementation of Sass compiler through gulp.

In order to compile the stylesheets you need to run this command from the repo root:

```
gulp sass
```

While developing it is also possible to run a watcher that compiles Sass files on change:

```
gulp
```

By default, source maps are not included in the compiled files. In order to turn them on while developing just add the `--debug` option:

```
gulp --debug
```

Icons

We are using gulp-iconfont to generate icon web fonts into `cms/static/cms/fonts/`. This also creates `_iconography.scss` within `cms/static/cms/sass/components` which adds all the icon classes and ultimately compiles to CSS.

In order to compile the web font you need to run:

```
gulp icons
```

This simply takes all SVGs within `cms/static/cms/fonts/src` and embeds them into the web font. All classes will be automatically added to `_iconography.scss` as previously mentioned.

Additionally we created an SVG template within `cms/static/cms/font/src/_template.svgz` that you should use when converting or creating additional icons. It is named *svgz* so it doesn't get compiled into the font. When using *Adobe Illustrator* please mind the following settings.

JS Bundling

JavaScript files are split up for easier development, but in the end they are bundled together and minified to decrease amount of requests made and improve performance. In order to do that we use the `gulp` task runner, where `bundle` command is available. We use Webpack for bundling JavaScript files. Configuration for each bundle are stored inside the `webpack.config.js` and their respective entry points. CMS exposes only one global variable, named `CMS`. If you want to use JavaScript code provided by CMS in external applications, you can only use bundles distributed by CMS, not the source modules.

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.

Contributions to the documentation earn the greatest respect from the core developers and the django CMS community.

Documentation should be:

- written using valid Sphinx/restructuredText syntax (see below for specifics); the file extension should be `.rst`
- wrapped at 100 characters per line
- written in English, using British English spelling and punctuation
- 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

Merging documentation is pretty fast and painless.

Except for the tiniest of change, we recommend that you test them before submitting.

Building the documentation

Follow the same steps above to fork and clone the project locally. Next, `cd` into the `django-cms/docs` and install the requirements:

```
make install
```

Now you can test and run the documentation locally using:

```
make run
```

This allows you to review your changes in your local browser using `http://localhost:8001/`.

Note: What this does

`make install` is roughly the equivalent of:

```
virtualenv env
source env/bin/activate
pip install -r requirements.txt
cd docs
make html
```

`make run` runs `make html`, and serves the built documentation on port 8001 (that is, at `http://localhost:8001/`.

It then watches the `docs` directory; when it spots changes, it will automatically rebuild the documentation, and refresh the page in your browser.

Spelling

We use sphinxcontrib-spelling, which in turn uses pyenchant and enchant to check the spelling of the documentation.

You need to check your spelling before submitting documentation.

Important: We use British English rather than US English spellings. This means that we use *colour* rather than *color*, *emphasise* rather than *emphasize* and so on.

Install the spelling software

`sphinxcontrib-spelling` and `pyenchant` are Python packages that will be installed in the virtualenv `docs/env` when you run `make install` (see above).

You will need to have `enchant` installed too, if it is not already. The easy way to check is to run `make spelling` from the `docs` directory. If it runs successfully, you don't need to do anything, but if not you will have to install `enchant` for your system. For example, on OS X:

```
brew install enchant
```

or Debian Linux:

```
apt-get install enchant
```

Check spelling

Run:

```
make spelling
```

in the `docs` directory to conduct the checks.

Note: This script expects to find a virtualenv at `docs/env`, as installed by `make install` (see above).

If no spelling errors have been detected, `make spelling` will report:

```
build succeeded.
```

Otherwise:

```
build finished with problems.
make: *** [spelling] Error 1
```

It will list any errors in your shell. Misspelt words will be also be listed in `build/spelling/output.txt`

Words that are not in the built-in dictionary can be added to `docs/spelling_wordlist`. **If** you are certain that a word is incorrectly flagged as misspelt, add it to the `spelling_wordlist` document, in alphabetical order. **Please do not add new words unless you are sure they should be in there.**

If you find technical terms are being flagged, please check that you have capitalised them correctly - `javascript` and `css` are **incorrect** spellings for example. Commands and special names (of classes, modules, etc) in double backticks - ``` `` - will mean that they are not caught by the spelling checker.

Important: You may well find that some words that pass the spelling test on one system but not on another. Dictionaries on different systems contain different words and even behave differently. The important thing is that the spelling tests pass on Travis when you submit a pull request.

Making a pull request

Before you commit any changes, you need to check spellings with `make spelling` and 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 in the usual way.

Documentation structure

Our documentation is divided into the following main sections:

- *Tutorials* (`introduction`): step-by-step, beginning-to-end tutorials to get you up and running
- *How-to guides* (`how_to`): step-by-step 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`)
- *Using django CMS* (`user`): guides for *using* rather than setting up or developing for the CMS

Documentation markup

Sections

We mostly follow the Python documentation conventions for section marking:

```
##########
Page title
##########

*******
heading
*******

sub-heading
===========

sub-sub-heading
---------------

sub-sub-sub-heading
^^^^^^^^^^^^^^^^^^^

sub-sub-sub-sub-heading
"""""""""""""""""""""""
```

Inline markup

- **use backticks - `` - for:**
  - literals:

    ```
    The ``cms.models.pagemodel`` contains several important methods.
    ```

  - filenames:

    ```
    Before you start, edit ``settings.py``.
    ```

  - names of fields and other specific items in the Admin interface:

    ```
    Edit the ``Redirect`` field.
    ```

- **use emphasis - *Home* - around:**
  - the names of available options in or parts of the Admin:

    ```
    To hide and show the *Toolbar*, use the...
    ```

  - the names of important modes or states:

    ```
    ... in order to switch to *Edit mode*.
    ```

  - values in or of fields:

    ```
    Enter *Home* in the field.
    ```

- **use strong emphasis - ** - around:**
  - buttons that perform an action:

```
Hit **View published** or **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

Create:

```
.. _testing:
```

and use:

```
:ref:`testing`
```

internal cross-references liberally.

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.

Contributing 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.

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 succinct 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).

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 license 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-X.Y.txt

# run the test suite
# note that you must be in the django-cms directory when you do this,
```

(continues on next page)

(continued from previous page)

```
# otherwise you'll get "Template not found" errors
cd django-cms
python manage.py test
```

It can take a few minutes to run.

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

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, `gettext` 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` 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:

```
manage.py test cms.PlaceholderBasicTests
```

and it should then run without errors.

Advanced testing options

Run `manage.py test --help` for the full list of advanced options.

Use `--parallel` to distribute the test cases across your CPU cores.

Use `--failed` to only run the tests that failed during the last run.

Use `--retest` to run the tests using the same configuration as the last run.

Use `--vanilla` to bypass the advanced testing system and use the built-in Django test command.

To use a different database, set the `DATABASE_URL` environment variable to a dj-database-url compatible value.

Running Frontend Tests

We have two types of frontend tests: unit tests and integration tests. For unit tests we are using Karma as a test runner and Jasmine as a test framework.

Integration tests run on PhantomJS and are built using CasperJS.

In order to be able to run them you need to install necessary dependencies as outlined in *frontend tooling installation instructions*.

Linting runs against the test files as well with `gulp tests:lint`. In order to run linting continuously, do:

```
gulp watch
```

Unit tests

Unit tests can be run like this:

```
gulp tests:unit
```

If your code is failing and you want to run only specific files, you can provide the `--tests` parameter with comma separated file names, like this:

```
gulp tests:unit --tests=cms.base,cms.modal
```

If you want to run tests continuously you can use the watch command:

```
gulp tests:unit:watch
```

This will rerun the suite whenever source or test file is changed. By default the tests are running on PhantomJS, but when running Karma in watch mode you can also visit the server it spawns with an actual browser.

> INFO [karma]: Karma v0.13.15 server started at http://localhost:9876/

On Travis CI we are using SauceLabs integration to run tests in a set of different real browsers, but you can opt out of running them on saucelabs using `[skip saucelabs]` marker in the commit message, similar to how you would skip the build entirely using `[skip ci]`.

We're using Jasmine as a test framework and Istanbul as a code coverage tool.

Integration tests

In order to run integration tests you'll have to install at least the version of django CMS from the current directory and djangocms-helper into into your virtualenv. All commands should be run from the root of the repository. If you do not have virtualenv yet, create and activate it first:

```
virtualenv env
. env/bin/activate
```

Then install minimum required dependencies:

```
pip install -r test_requirements/django-1.8.txt
pip install -e .
```

Now you'll be able to run a tests with this command:

```
gulp tests:integration
```

The command will start a server, wait for a minute for the migrations to run and will run integration tests against it. It will use `testdb.sqlite` as the database. If you want to start with a clean state you could use `--clean` argument.

Some tests require different server configuration, so it is possible that the server will stop, and another variation will start with different arguments. Take a look inside *testserver.py* if you need to customise the test server settings.

While debugging you can use the `--tests` parameter as well in order to run test suites separately.:

```
gulp tests:integration --tests=pagetree
gulp tests:integration --tests=loginAdmin,toolbar
```

If specified tests require different servers they will be grouped to speed things up, so the order might not be the same as you specify in the argument.

When running locally, it sometimes helps to visualise the tests output. For that you can install casper-summoner utility (`npm install -g casper-summoner`), and run the tests with additional `--screenshots` argument. It will create `screenshots` folder with screenshots of almost every step of each test. Subsequent runs will override the existing files. Note that this is experimental and may change in the future.

It might sometimes be useful not to restart the server when creating the tests, for that you can run `python testserver.py` with necessary arguments in one shell and `gulp tests:integration --no-server` in another. However you would need to clean the state yourself if the test you've been writing fails.

Writing tests

Contributing tests is widely regarded as a very prestigious contribution (you're making everybody's future work much easier by doing so). We'll always accept contributions of a test without code, but not code without a test - which should give you an idea of how important tests are.

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.

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.

4.1.6 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.

3.4.7 release notes

What's new in 3.4.7

Bug Fixes

- Removed extra quotation mark from the sideframe button template
- Fixed a bug where xframe options were processed by clickjacking middleware when page was served from cache, rather then get this value from cache
- Fixed a bug where cached page permissions overrides global permissions
- Fixed a bug where editing pages with primary keys greater than 9999 would throw an exception.
- Fixed broken wizard page creation when no language is set within the template context (see #5828).
- Fixed a security vulnerability in the plugin_type url parameter to insert JavaScript code.

3.4.6 release notes

What's new in 3.4.6

Bug Fixes

- Changed the way drag and drop works in the page tree. The page has to be selected first before moving.
- Fixed a bug where the cms alias plugin leaks context into the rendered aliased plugins.
- Fixed a bug where users without the "Change advanced settings" permission could still change a page's template.
- Added `on_delete` to `ForeignKey` and `OneToOneField` to silence Django deprecation warnings.
- Fixed a bug where the sitemap would ignore the `public` setting of the site languages and thus display hidden languages.

- Fixed an `AttributeError` raised when adding or removing apphooks in Django 1.11.
- Fixed an `InconsistentMigrationHistory` error raised when the contenttypes app has a pending migration after the user has applied the `0010_migrate_use_structure` migration.

3.4.5 release notes

This version of django CMS is the first to introduce compatibility with Django 1.11, itself also a Long-Term Support release.

What's new in 3.4.5

Bug Fixes

- Fixed a bug where slug wouldn't be generated in the creation wizard
- Fixed a bug where the add page endpoint rendered `Change page` as the html title.
- Fixed an issue where non-staff users could request the wizard create endpoint.
- Fixed an issue where the `Edit page` toolbar button wouldn't show on non-cms pages with placeholders.
- Fixed a bug where placeholder inheritance wouldn't work if the inherited placeholder is cached in an ancestor page.
- Fixed a regression where the code following a `{% placeholder x or %}` declaration, was rendered before attempting to inherit content from parent pages.
- Changed page/placeholder cache keys to use sha1 hash instead of md5 to be FIPS compliant.
- Fixed a bug where the change of a slug would not propagate to all descendant pages
- Fixed a `ValueError` raised when using `ManifestStaticFilesStorage` or similar for static files. This only affects Django >= 1.10

Improvements and new features

- Introduced Django 1.11 compatibility

3.4.4 release notes

What's new in 3.4.4

Bug Fixes

- Fixed a bug in which cancelling the publishing dialog wasn't respected.
- Fixed a bug causing post-login redirection to an incorrect URL on single-language sites.
- Fixed an error when retrieving placeholder label from configuration.
- Fixed a bug which caused certain translations to display double-escaped text in the page list admin view.
- Adjusted the toolbar JavaScript template to escape values coming from the request.
- Replaced all custom markup on the `admin/cms/page/includes/fieldset.html` template with an `{% include %}` call to Django's built-in `fieldset.html` template.

- Fixed a bug which prevented a page from being marked as dirty when a placeholder was cleared.
- Fixed an IntegrityError raised when publishing a page with no public version and whose publisher state was pending.
- Fixed an issue with JavaScript not being able to determine correct path to the async bundle
- Fixed a `DoesNotExist` database error raised when moving a page marked as published, but whose public translation did not exist.
- Fixed a bug in which the menu rendered nodes using the site session variable (set in the admin), instead of the current request site.
- Fixed a race condition bug in which the database cache keys were deleted without syncing with the cache server, and as a result old menu items would continue to be displayed.
- Fixed a 404 raised when using the `Delete` button for a Page or Title extension on Django >= 1.9
- Fixed a performance issue with nested pages when using the `inherit` flag on the `{% placeholder %}` tag.
- Fixed a bug in which the placeholder cache was not consistently cleared when a page was published.
- Fixed a regression which prevented users from setting a redirect to the homepage.

Improvements and new features

- Enhanced the plugin menu to not show plugins the user does not have permission to add.
- Added Dropdown class to toolbar items.
- Added "How to serve multiple languages" section to documentation.

Backwards incompatible changes

CMSPluginBase class

- Changed the signature for internal `cms.plugin_base.CMSPluginBase` methods `get_child_classes` and `get_parent_classes` to take an optional `instance` parameter.

Page model

The following methods have been removed from the `Page` model:

- `reset_to_live` This internal method was removed and replaced with `revert_to_live`.

Placeholder utilities

Because of a performance issue with placeholder inheritance, we've altered the return value for the following internal placeholder utility functions:

- `cms.utils.placeholder._scan_placeholders` This will now return a list of `Placeholder` tag instances instead of a list of placeholder slot names. You can get the slot name by calling the `get_name()` method on the `Placeholder` tag instance.

- `cms.utils.placeholder.get_placeholders` This will now return a list of `DeclaredPlaceholder` instances instead of a list of placeholder slot names. You can get the slot name by accessing the `slot` attribute on the `DeclaredPlaceholder` instance.

3.4.3 release notes

What's new in 3.4.3

Security Fixes

- Fixed a security vulnerability in the page redirect field which allowed users to insert JavaScript code.
- Fixed a security vulnerability where the `next` parameter for the toolbar login was not sanitised and could point to another domain.

Thanks

Thanks to Mark Walker and Anthony Steinhauser for reporting the security issues.

3.4.2 release notes

django CMS 3.4.2 introduces two key new features: *Revert to live* for pages, and support for Django 1.10

Revert to live is in fact being reintroduced in a new form following a complete rewrite of our revision handling system, that was removed in *django CMS 3.4* to make possible a greatly-improved new implementation from scratch.

Revert to live is the first step in fully re-implementing revision management on a new basis.

The full set of changes is listed below.

What's new in 3.4.2

Bug Fixes

- Escaped strings in `close_frame` JS template.
- Fixed a bug with *text-transform* styles on inputs affecting CMS login
- Fixed a typo in the confirmation message for copying plugins from a different language
- Fixed a bug which prevented certain migrations from running in a multi-db setup.
- Fixed a regression which prevented the `Page` model from rendering correctly when used in a `raw_id_field`.
- Fixed a regression which caused the CMS to cache the toolbar when `CMS_PAGE_CACHE` was set to `True` and an anonymous user had `cms_edit` set to `True` on their session.
- Fixed a regression which prevented users from overriding content in an inherited placeholder.
- Fixed a bug affecting Firefox for Macintosh users, in which use of the Command key later followed by Return would trigger a plugin save.
- Fixed a bug where template inheritance setting creates spurious migration (see #3479)
- Fixed a bug which prevented the page from being marked as dirty (pending changes) when changing the value of the overwrite url field.

- Fixed a bug where the page tree would not update correctly when a sibling page was moved from left to right or right to left.

Improvements and new features

- Added official support for Django 1.10.
- Rewrote manual installation how-to documentation
- Re-introduced the "Revert to live" menu option.
- Added support for django-reversion >= 2 (see #5830)
- Improved the `fix-tree` command so that it also fixes non-root nodes (pages).
- Introduced placeholder operation signals.

Deprecations

- Removed the deprecated `add_url()`, `edit_url()`, `move_url()`, `delete_url()`, `copy_url()` properties of CMSPlugin model.
- Added a deprecation warning to method `render_plugin()` in class `CMSPlugin`.
- Deprecated `frontend_edit_template` attribute of `CMSPluginBase`.
- The `post_` methods in `` `PlaceholderAdminMixin `` have been deprecated in favor of placeholder operation signals.

Other changes

- Adjusted Ajax calls triggered when performing a placeholder operation (add plugin, etc..) to include a GET query called cms_path. This query points to the path where the operation originates from.
- Changed CMSPlugin.get_parent_classes() from method to classmethod.

3.4.1 release notes

What's new in 3.4.1

Bug Fixes

- Fixed a regression when static placeholder was uneditable if it was present on the page multiple times
- Removed globally unique constraint for Apphook configs.
- Fixed a bug when keyboard shortcuts were triggered when form fields were focused
- Fixed a bug when `shift + space` shortcut wouldn't correctly highlight a plugin in the structure board
- Fixed a bug when plugins that have top-level svg element would break structure board
- Fixed a bug where output from the `show_admin_menu_for_pages` template tag was escaped in Django 1.9
- Fixed a bug where plugins would be rendered as editable if toolbar was shown but user was not in edit mode.
- Fixed CSS reset issue with shortcuts modal

3.4 release notes

The most significant change in this release is the removal of revision support (i.e. undo/redo/recover functionality on pages) from the core django CMS. This functionality will be reinstated as an optional addon in due course, but in the meantime, that functionality is not available.

What's new in 3.4

- Changed the way CMS plugins are rendered. The HTML `div` with `cms-plugin` class is no longer rendered around every CMS plugin. Instead a combination of `template` tags and JavaScript is used to add event handlers and plugin data directly to the plugin markup. This fixes most of the rendering issues caused by the extra markup.
- Changed asset cache-busting implementation, which is now handled by a path change, rather than the `GET` parameter.
- Added the option to copy pages in the page tree using the drag and drop interface.
- Made it possible to use multi-table inheritance for Page/Title extensions.
- Refactored plugin rendering functionality to speed up loading time in both structure and content modes.
- Added a new `Shift` + `Space` shortcut to switch between structure and content mode while highlighting the current plugin, revealing its position.
- Improved keyboard navigation
- Added help modal about available shortcuts
- Added fuzzy matching to the plugin picker.
- Changed the `downcast_plugins` utility to return a generator instead of a list.
- Fixed a bug that caused an aliased placeholder to show in structure mode.
- Fixed a bug that prevented aliased content from showing correctly without publishing the page first.
- Added help text to an `Alias` plugin change form when attached to a page to show the content editor where the content is aliased from.
- Removed revision support from django CMS core. As a result both `CMS_MAX_PAGE_HISTORY_REVERSIONS` and `CMS_MAX_PAGE_PUBLISH_REVERSIONS` settings are no longer supported, as well as the `with_revision` parameter in `cms.api.create_page` and `cms.api.create_title`.
- In `cms.plugin_base.CMSPluginBase` methods `get_child_classes` and `get_parent_classes` now are implemented as a `@classmethod`.

Upgrading to 3.4

A database migration is required because the default value of CMSPlugin.position was set to 0 instead of null.

Please make sure that your current database is consistent and in a healthy state, and **make a copy of the database before proceeding further.**

Then run:

```
python manage.py migrate
python manage.py cms fix-tree
```

Backward incompatible changes

Apphooks & Toolbars

As per our deprecation policy we've now removed the backwards compatible shim for `cms_app.py` and `cms_toolbar.py`. If you have not done so already, please rename these to `cms_apps.py` and `cms_toolbars.py`.

Permissions

The permissions system was heavily refactored. As a result, several internal functions and methods have been removed or changed.

Functions removed:

- `user_has_page_add_perm`
- `has_page_add_permission`
- `has_page_add_permission_from_request`
- `has_any_page_change_permissions`
- `has_auth_page_permission`
- `has_page_change_permission`
- `has_global_page_permission`
- `has_global_change_permissions_permission`
- `has_generic_permission`
- `load_view_restrictions`
- `get_any_page_view_permissions`

The following methods were changed to require a user parameter instead of a request:

- `Page.has_view_permission`
- `Page.has_add_permission`
- `Page.has_change_permission`
- `Page.has_delete_permission`
- `Page.has_delete_translation_permission`
- `Page.has_publish_permission`
- `Page.has_advanced_settings_permission`
- `Page.has_change_permissions_permission`
- `Page.has_move_page_permission`

These are also deprecated in favor of their counterparts in `cms.utils.page_permissions`.

To keep consistency with both django CMS permissions and Django permissions, we've modified the vanilla permissions system (`CMS_PERMISSIONS = False`) to require users to have certain Django permissions to perform an action.

Here's an overview:

Action	Permission required
Add Page	Can Add Page & Can Change Page
Change Page	Can Change Page
Delete Page	Can Change Page & Can Delete Page
Move Page	Can Change Page
Publish Page	Can Change Page & Can Publish Page

This change will only affect non-superuser staff members.

Warning: If you have a custom `Page` extension with a configured toolbar, please see the updated *example*. It uses the new permission internals.

Manual plugin rendering

We've rewritten the way plugins and placeholders are rendered. As a result, if you're manually rendering plugins and placeholders you'll have to adapt your code to match the new rendering mechanism.

To render a plugin programmatically, you will need a context and request object.

Warning: Manual plugin rendering is not a public API, and as such it's subject to change without notice.

```
from django.template import RequestContext
from cms.plugin_rendering import ContentRenderer

def render_plugin(request, plugin):
    renderer = ContentRenderer(request)
    context = RequestContext(request)
    # Avoid errors if plugin require a request object
    # when rendering.
    context['request'] = request
    return renderer.render_plugin(plugin, context)
```

Like a plugin, to render a placeholder programmatically, you will need a context and request object.

Warning: Manual placeholder rendering is not a public API, and as such it's subject to change without notice.

```
from django.template import RequestContext
from cms.plugin_rendering import ContentRenderer

def render_placeholder(request, placeholder):
    renderer = ContentRenderer(request)
    context = RequestContext(request)
    # Avoid errors if plugin require a request object
    # when rendering.
    context['request'] = request
    content = renderer.render_placeholder(
       placeholder,
       context=context,
    )
    return content
```

3.3 release notes

django CMS 3.3 has been planned largely as a consolidation release, to build on the progress made in 3.2 and pave the way for the future ones.

The largest major change is dropped support for Django 1.6 and 1.7, and Python 2.6 followed by major code cleanup to remove compatibility shims.

What's new in 3.3

- Removed support for Django 1.6, 1.7 and python 2.6
- Changed the default value of CMSPlugin.position to 0 instead of null
- Refactored the language menu to allow for better integration with many languages
- Refactored management commands completely for better consistency
- Fixed "failed to load resource" for favicon on welcome screen
- Changed behaviour of toolbar CSS classes: `cms-toolbar-expanded` class is only added now when toolbar is fully expanded and not at the beginning of the animation. `cms-toolbar-expanding` and `cms-toolbar-collapsing` classes are added at the beginning of their respective animations.
- Added unit tests for CMS JavaScript files
- Added frontend integration tests (written with Casper JS)
- Removed frontend integration tests (written with Selenium)
- Added the ability to declare cache expiration periods on a per-plugin basis
- Improved UI of page tree
- Improved UI in various minor ways
- Added a new setting CMS_INTERNAL_IPS for defining a set of IP addresses for which the toolbar will appear for authorized users. If left unset, retains the existing behavior of allowing toolbar for authorized users at any IP address.
- Changed behaviour of sideframe; is no longer resizable, opens to 90% of the screen or 100% on small screens.
- Removed some unnecessary reloads after closing sideframe.
- Added the ability to make pagetree actions work on currently picked language
- Removed deprecated CMS_TOOLBAR_SIMPLE_STRUCTURE_MODE setting
- Introduced the method `get_cache_expiration` on CMSPluginBase to be used by plugins for declaring their rendered content's period of validity.
- Introduced the method `get_vary_cache_on` on CMSPluginBase to be used by plugins for declaring `VARY` headers.
- Improved performance of plugin moving; no longer saves all plugins inside the placeholder.
- Fixed breadcrumbs of recently moved plugin reflecting previous position in the tree
- Refactored plugin adding logic to no longer create the plugin before the user submits the form.
- Improved the behaviour of the placeholder cache
- Improved fix-tree command to sort by position and path when rebuilding positions.
- Fixed several regressions and tree corruptions on page move.

- Added new class method on CMSPlugin `requires_parent_plugin`
- Fixed behaviour of `get_child_classes`; now correctly calculates child classes when not configured in the placeholder.
- Removed internal `ExtraMenuItems` tag.
- Removed internal `PluginChildClasses` tag.
- Modified RenderPlugin tag; no longer renders the `content.html` template and instead just returns the results.
- Added a `get_cached_template` method to the `Toolbar()` main class to reuse loaded templates per request. It works like Django's cached template loader, but on a request basis.
- Added a new method `get_urls()` on the appbase class to get CMSApp.urls, to allow passing a page object to it.
- Changed JavaScript linting from JSHint and JSCS to ESLint
- Fixed a bug when it was possible to drag plugin into clipboard
- Fixed a bug where clearing clipboard was closing any open modal
- Added CMS_WIZARD_CONTENT_PLACEHOLDER setting
- Renamed the CMS_WIZARD_* settings to CMS_PAGE_WIZARD_*
- Deprecated the old-style wizard-related settings
- Improved documentation further
- Improved handling of uninstalled apphooks
- Fixed toolbar placement when foundation is installed
- Fixed an issue which could lead to an apphook without a slug
- Fixed numerous frontend issues
- Added contribution policies documentation
- Corrected an issue where someone could see and use the internal placeholder plugin in the structure board
- Fixed a regression where the first page created was not automatically published
- Corrected the instructions for using the `delete-orphaned-plugins` command
- Re-pinned django-treebeard to >=4.0.1

Upgrading to 3.3

A database migration is required because the default value of CMSPlugin.position was set to 0 instead of null.

Please make sure that your current database is consistent and in a healthy state, and **make a copy of the database before proceeding further.**

Then run:

```
python manage.py migrate
python manage.py cms fix-tree
```

Deprecation of Old-Style Page Wizard Settings

In this release, we introduce a new naming scheme for the Page Wizard settings that better reflects that they effect the CMS's Page Wizards, rather than all wizards. This will also allow future settings for different wizards with a smaller chance of confusion or naming-collision.

This release simultaneously deprecates the old naming scheme for these settings. Support for the old naming scheme will be dropped in version 3.5.0.

Action Required

Developers using any of the following settings in their projects should rename them as follows at their earliest convenience.

```
CMS_WIZARD_DEFAULT_TEMPLATE => CMS_PAGE_WIZARD_DEFAULT_TEMPLATE
CMS_WIZARD_CONTENT_PLUGIN => CMS_PAGE_WIZARD_CONTENT_PLUGIN
CMS_WIZARD_CONTENT_PLUGIN_BODY => CMS_PAGE_WIZARD_CONTENT_PLUGIN_BODY
CMS_WIZARD_CONTENT_PLACEHOLDER => CMS_PAGE_WIZARD_CONTENT_PLACEHOLDER
```

The CMS will accept both-schemes until 3.5.0 when support for the old scheme will be dropped. During this transition period, the CMS prefers the new-style naming if both schemes are used in a project's settings.

Backward incompatible changes

Management commands

Management commands uses now argparse instead of optparse, following the Django deprecation of the latter API.

The commands behaviour has remained untouched.

Detailed changes:

- commands now use argparse subcommand API which leads to slightly different help output and other internal differences. If you use the commands by using Django's `call_command` function you will have to adapt the command invocation to reflect this.
- some commands have been rename replacing underscores with hyphens for consistency
- all arguments are now non-positional. If you use the commands by using Django's `call_command` function you will have to adapt the command invocation to reflect this.

Signature changes

The signatures of the toolbar methods `get_or_create_menu` have a new kwarg `disabled` *inserted* (not appended). This was done to maintain consistency with other, existing toolbar methods. The signatures are now:

- `cms.toolbar.items.Menu.get_or_create_menu(key, verbose_name, disabled=False, side=LEFT, position=None)`
- `cms.toolbar.toolbar.CMSToolbar.get_or_create_menu(key, verbose_name=None, disabled=False, side=LEFT, position=None)`

It should only affect developers who use kwargs as positional args.

3.2.5 release notes

What's new in 3.2.5

Note: This release is identical to 3.2.4, but had to be released also as 3.2.4 due to a Python wheel packaging issue.

Bug Fixes

- Fix cache settings
- Fix user lookup for view restrictions/page permissions when using raw id field
- Fixed regression when page couldn't be copied if CMS_PERMISSION was False
- Fixes an issue relating to uninstalling a namespaced application
- Adds "Can change page" permission
- Fixes a number of page-tree issues the could lead data corruption under certain conditions
- 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 un-escaped, 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 3.2.1 is not compatible with djangocms-text-ckeditor < 2.8.1. If you're using djangocms-text-ckeditor, please upgrade to 2.8.1 or later.

3.2.4 release notes

What's new in 3.2.4

Bug Fixes

- Fix cache settings
- Fix user lookup for view restrictions/page permissions when using raw id field
- Fixed regression when page couldn't be copied if CMS_PERMISSION was False
- Fixes an issue relating to uninstalling a namespaced application
- Adds "Can change page" permission
- Fixes a number of page-tree issues the could lead data corruption under certain conditions
- 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 3.2.1 is not compatible with djangocms-text-ckeditor < 2.8.1. If you're using djangocms-text-ckeditor, please upgrade to 2.8.1 or later.

3.2.3 release notes

What's new in 3.2.3

Bug Fixes

- Fix the display of hyphenated language codes in the page tree
- Fix a family of issues relating to unescaped translations in the page tree

3.2.2 release notes

What's new in 3.2.2

Improvements

- Substantial "under-the-hood" improvements to the page tree resulting in significant reduction of page-tree reloads and generally cleaner code
- Update jsTree version to 3.2.1 with slight adaptations to the page tree
- Improve the display and usability of the language menu, especially in cases where there are many languages
- Documentation improvements

Bug Fixes

- Fix an issue relating to search fields in plugins
- Fix an issue where the app-resolver would trigger locales into migrations
- Fix cache settings
- Fix ToolbarMiddleware.is_cms_request logic
- Fix numerous Django 1.9 deprecations
- Numerous other improvements to overall stability and code quality

Model Relationship Back-References and Django 1.9

Django 1.9 is lot stricter about collisions in the `related_names` of relationship fields than previous versions of Django. This has brought to light issues in django CMS relating to the private field `CMSPlugin.cmsplugin_ptr`. The issue becomes apparent when multiple packages are installed that provide plugins with the same model class name. A good example would be if you have the package `djangocms-file` installed, which has a poorly named CMSPlugin model subclass called `File`, then any other package that has a plugin with a field named "file" would most likely cause an issue. Considering that `djangocms-file` is a very common plugin to use and a field name of "file" is not uncommon in other plugins, this is less than ideal.

Fortunately, developers can correct these issues in their own projects while they await improvements in django CMS. There is an internal field that is created when instantiating plugins: `CMSPlugin.cmsplugin_ptr`. This private field is declared in the CMSPlugin base class and is populated on instantiation using the lower-cased model name of the CMSPlugin subclass that is being registered.

A subclass to `CMSPlugin` can declare their own `cmsplugin_ptr` field to immediately fix this issue. The easiest solution is to declare this field with a `related_name` of "+". In typical Django fashion, this will suppress the back-reference and prevent any collisions. However, if the back-reference is required for some reason (very rare), then we recommend using the pattern `%(app_label)s_%(class_name)s`. In fact, in version 3.3 of django CMS, this is precisely the string-template that the reference setup will use to create the name. Here's an example:

```
class MyPlugin(CMSPlugin):
    class Meta:
        app_label = 'my_package'

    cmsplugin_ptr = models.OneToOneField(
        CMSPlugin,
        related_name='my_package_my_plugin',
        parent_link=True
    )

    # other fields, etc.
    # ...
```

Please note that CMSPlugin.cmsplugin_ptr will remain a private field.

Notice of Upcoming Change in 3.3

As outlined in the section immediately above, the pattern currently used to derive a `related_name` for the private field `CMSPlugin.cmsplugin_ptr` may result in frequent collisions. In django CMS 3.3, this string-template will be changed to utilise both the `app_label` and the model class name. In the majority of cases, this will not affect developers or users, but if your project uses these back-references for some reason, please be aware of this change and plan accordingly.

Treebeard corruption

Prior to 3.2.1 moving or pasting nested plugins could lead to some non-fatal tree corruptions, raising an error when adding plugins under the newly pasted plugins.

To fix these problems, upgrade to 3.2.1 or later and then run `manage.py cms fix-tree` command to repair the tree.

DjangoCMS Text CKEditor

Action required

CMS 3.2.2 is not compatible with djangocms-text-ckeditor < 2.8.1. If you're using djangocms-text-ckeditor, please upgrade to 2.8.1 or up.

3.2.1 release notes

What's new in 3.2.1

Improvements

- Add support for Django 1.9 (with some deprecation warnings).
- Add support for django-reversion 1.10+ (required by Django 1.9+).
- Add placeholder name to the edit tooltip.
- Add `attr['is_page']=True` to CMS Page navigation nodes.

- Add Django and Python versions to debug bar info tooltip

Bug Fixes

- Fix an issue with refreshing the UI when switching CMS language.
- Fix an issue with sideframe urls not being remembered after reload.
- Fix breadcrumb in page revision list.
- Fix clash with Foundation that caused "Add plugin" button to be unusable.
- Fix a tree corruption when pasting a nested plugin under another plugin.
- Fix message with CMS version not showing up on hover in debug mode.
- Fix messages not being positioned correctly in debug mode.
- Fix an issue where plugin parent restrictions where not respected when pasting a plugin.
- Fix an issue where "Copy all" menu item could have been clicked on empty placeholder.
- Fix a bug where page tree styles didn't load from STATIC_URL that pointed to a different host.
- Fix an issue where the side-frame wouldn't refresh under some circumstances.
- Honour CMS_RAW_ID_USERS in GlobalPagePermissionAdmin.

Treebeard corruption

Prior to 3.2.1 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 3.2.1 and then run `manage.py cms fix-tree` command to repair the tree.

DjangoCMS Text CKEditor

Action required

CMS 3.2.1 is not compatible with djangocms-text-ckeditor < 2.8.1. If you're using djangocms-text-ckeditor, please upgrade to 2.8.1 or up.

3.2 release notes

django CMS 3.2 introduces touch-screen support, significant improvements to the structure-board, and numerous other updates and fixes for the frontend. Behind the scenes, auto-reloading following apphook configuration changes will make life simpler for all users.

> **Warning:** Upgrading from previous versions
>
> 3.2 introduces some changes that **require** action if you are upgrading from a previous version. Please read *Upgrading django CMS 3.1 to 3.2* for a step-by-step guide to the process of upgrading from 3.1 to 3.2.

What's new in 3.2

- *new welcome page* to help new users
- touch-screen support for most editing interfaces, for sizes from small tablets to table-top devices
- enhanced and polished user interface
- much-needed improvements to the structure-board
- enhancements to components such as the pop-up plugin editor, sideframe (now called the *overlay*) and the toolbar
- significant speed improvements on loading, HTTP requests and file sizes
- restarts are no longer required when changing apphook configurations
- a new content wizard system, adaptable to arbitrary content types

Changes that require attention

Touch interface support

For general information about touch interface support, see the *touch screen device notes* in the documentation.

Important: These notes about touch interface support apply only to the **django CMS admin and editing interfaces**. The visitor-facing published site is **wholly independent** of this, and the responsibility of the site developer. A good site should already work well for its visitors, whatever interface they use!

Numerous aspects of the CMS and its interface have been updated to work well with touch-screen devices. There are some restrictions and warnings that need to be borne in mind.

Device support

Smaller devices such as most phones are too small to be adequately usable. For example, your Apple Watch is sadly unlikely to provide a very good django CMS editing experience.

Older devices will often lack the performance to support a usefully responsive frontend editing/administration interface.

There are some device-specific issues still to be resolved. Some of these relate to the CKEditor (the default django CMS text editor). We will continue to work on these and they will be addressed in a future release.

See *Device support* for information about devices that have been tested and confirmed to work well, and about known issues affecting touch-screen device support.

Feedback required

We've tested the CMS interface extensively, but will be very keen to have feedback from other users - device reports, bug reports and general suggestions and opinions are very welcome.

Bug-fixes

- An issue in which `{% placeholder %}` template tags ignored the `lang` parameter has been fixed.

  However this may affect the behaviour of your templates, as now a previously-ignored parameter will be recognised. If you used the `lang` parameter in these template tags you may be affected: check the behaviour of your templates after upgrading.

Content wizards

Content creation wizards can help simplify production of content, and can be created to handle non-CMS content too.

For a quick introduction to using a wizard as a content editor, see the *user tutorial*.

Renaming cms_app, cms_toolbar, menu modules

`cms_app.py`, `cms_toolbar.py` and `menu.py` have been renamed to `cms_apps.py`, `cms_toolbars.py` and `cms_menus.py` for consistency.

Old names are still supported but deprecated; support will be removed in 3.4.

Action required

In your own applications that use these modules, rename `cms_app.py` to `cms_apps.py`, `cms_toolbar.py` to `cms_toolbars.py` and `menu.py` to `cms_menus.py`.

New `ApphookReloadMiddleware`

Until now, changes to apphooks have required a restart of the server in order to take effect. A new optional middleware class, *cms.middleware.utils.ApphookReloadMiddleware*, makes this automatic.

For developers

Various improvements have been implemented to make developing with and for django CMS easier. These include:

- improvements to frontend code, to comply better with aldryn-boilerplate-bootstrap3
- changes to directory structure for frontend related components such as JavaScript and SASS.
- We no longer use `develop.py`; we now use `manage.py` for all development tasks. See *How to contribute a patch* for examples.
- We've moved our `widgets.py` JavaScript to `static/cms/js/widgets`.

Code formatting

We've switched from tabs (in some places) to four spaces *everywhere*. See *Contributing code* for more on formatting.

gulp.js

We now use *gulp.js* for linting, compressing and bundling of frontend files.

Sass-related changes

We now use LibSass rather than Compass for building static files (this only affects frontend developers *of* django CMS - contributors to it, not other users or developers). We've also adopted CSSComb.

`.editorconfig` file

We've added a `.editorconfig` (at the root of the project) to provide cues to text editors.

Automated spelling checks for documentation

Documentation is now checked for spelling. A `make spelling` command is available now when working on documentation, and our Travis Continuous Integration server also runs these checks.

See the *Spelling* section in the documentation.

New structure board

The structure board is cleaner and easier to understand. It now displays its elements in a tree, rather than in a series of nested boxes.

You can optionally enable the old appearance and behaviour with the `CMS_TOOLBAR_SIMPLE_STRUCTURE_MODE` setting (this option will be removed in 3.3).

Replaced the sideframe with an overlay

The sideframe that could be expanded and collapsed to reveal a view of the admin and other controls has been replaced by a simpler and more elegant *overlay* mechanism.

The API documentation still refers to the `sideframe`, because it is invoked in the same way, and what has changed is merely the behaviour in the user's browser.

In other words, *sideframe* and the *overlay* refer to different versions of the same thing.

New startup page

A new startup mode makes it easier for users (especially new users) to dive straight into editing when launching a new site. See the *Tutorial* for more.

Known issues

The sub-pages of a page with an apphook will be unreachable (`404 page not found`), due to internal URL resolution mechanisms in the CMS. Though it's unlikely that most users will need sub-pages of this kind (typically, an apphooked page will create its own sub-pages) this issue will be addressed in a forthcoming release.

Backward-incompatible changes

See the *Frontend code* documentation.

There are no other known backward-incompatible changes.

Upgrading django CMS 3.1 to 3.2

Please note any changes that require action above, and take action accordingly.

A database migration is required (a new model, `UrlconfRevision` has been added as part of the apphook reload mechanism):

Note also that any third-party applications you update may have their own migrations, so as always, before upgrading, please make sure that your current database is consistent and in a healthy state, and **make a copy of the database before proceeding further.**

Then run:

```
python manage.py migrate
```

to migrate.

Otherwise django CMS 3.2 represents a fairly easy upgrade path.

Pending deprecations

In django CMS 3.3:

> Django 1.6, 1.7 and Python 2.6 will no longer be supported. If you still using these versions, you are strongly encouraged to begin exploring the upgrade process to a newer version.
>
> The `CMS_TOOLBAR_SIMPLE_STRUCTURE_MODE` setting will be removed.

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

3.1.4 release notes

What's new in 3.1.4

Bug Fixes

- Fixed a problem in `0010_migrate_use_structure.py` that broke some migration paths to Django 1.8
- Fixed `fix_tree` command
- Removed some warnings for Django 1.9
- Fixed issue causing plugins to move when using scroll bar of plugin menu in Firefox & IE
- Fixed JavaScript error when using `PageSelectWidget`
- Fixed whitespace markup issues in draft mode
- Added plugin migrations layout detection in tests
- Fixed some treebeard corruption issues

Treebeard corruption

Prior to 3.1.4 deleting pages could lead to some non-fatal tree corruptions, raising an error when publishing, deleting, or moving pages.

To fix these problems, upgrade to 3.1.4 and then run `manage.py cms fix-tree` command to repair the tree.

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 data migration
- 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 pop-up dialogs
- Exclude `scssc` from manifest
- Remove unpublished pages from menu
- Remove page from menu items for performance reason
- Fix access to 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.

3.1.2 release notes

What's new in 3.1.2

Bug Fixes

- Fix placeholder cache invalidation under some circumstances
- Update translations

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 editor 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 template tags
- Fix an error in placeholder cache
- Fix get_language_from_request if POST and GET exists
- Minor documentation fixes
- Revert whitespace clean-up on flash player to fix it
- Correctly restore previous status of drag bars
- Fix an issue related to "Empty all" Placeholder feature
- Fix plugin sorting in Python 3
- 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 sub-menu
- Fix language chooser template

Potentially backward incompatible 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 behaviour 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, maq-nouch, nikolas, northben, olarcheveque, pa0lin082, peterfarrell, sam-m888, sephii, stefanw, timgraham, 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.

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 Pre-order 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 (Materialised 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 (specifically, 1.6.9 and later) and 1.7.

Warning: Django security support

Django 1.6 support is provided as an interim measure only. In accordance with the Django Project's security policies, 1.6 no longer receives security updates from the Django Project team. Projects running on Django 1.6 have known vulnerabilities, so you are advised to upgrade your installation to 1.7 or 1.8 as soon as possible.

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]`.

Changes to PlaceholderAdmin.add_plugin

Historically, when a plugin was added to django CMS, a POST request was made to the `PlaceholderAdmin.add_plugin` endpoint (and going back into very ancient history before `PlaceholderAdmin` existed, it was `PageAdmin.add_plugin`). This would create an instance of `CMSPlugin`, but not an instance of the actual plugin model itself. It would then let the user agent edit the created plugin, which when saved would put the database back in to a consistent state, with a plugin instance connected to the otherwise empty and meaningless `CMSPlugin`.

In some cases, "ghost plugins" would be created, if the process of creating the plugin instance failed or were interrupted, for example by the browser window's being closed.

This would leave orphaned `CMSPlugin` instances in the database without any data. This could result pages not working at all, due to the resulting database inconsistencies.

This issue has now been solved. Calling `CMSPluginBase.add_plugin` with a GET request now serves the form for creating a new instance of a plugin. Then on submitting that form via POST, the plugin is created in its entirety, ensuring a consistent database and an end to ghost plugins.

However, to solve it some backwards incompatible changes to **non-documented APIs** that developers might have used have had to be made.

CMSPluginBase permission hooks

Until now, `CMSPluginBase.has_delete_permission`, `CMSPluginBase.has_change_permission` and `CMSPluginBase.has_add_permission` were handled by a single method, which used an undocumented and unreliable property on `CMSPluginBase` instances (or subclasses thereof) to handle permission management.

In 3.1, `CMSPluginBase.has_add_permission` is its own method that implements proper permission checking for adding plugins.

If you want to work with those APIs, see the Django documentation for more on the permission methods.

CMSPluginBase.get_form

Prior to 3.1, this method would only ever be called with an actual instance available.

As of 3.1, this method will be called without an instance (the `obj` argument to the method will be `None`) if the form is used to add a plugin, rather than editing it. Again, this is in line with how Django's `ModelAdmin` works.

If you need access to the `Placeholder` object to which the plugin will be added, the `request` object is *guaranteed* to have a `placeholder_id` key in `request.GET`, which is the primary key of the `Placeholder` object to which the plugin will be added. Similarly, `plugin_language` in `request.GET` holds the language code of the plugin to be added.

CMSPlugin.add_view

This method used to never be called, but as of 3.1 it will be. Should you need to hook into this method, you may want to use the `CMSPluginBase.add_view_check_request` method to verify that a request made to this view is valid. This method will perform integrity and permission checks for the GET parameters of the request.

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 changed 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.

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 template tags

render_model_add_block

The family of *render_model* template tags 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 behaviour for placeholders.

Update the database

- Rename plugin table names, to conform to the new naming scheme (see above). **Be warned** that not all third-party plugin applications may provide these migrations - in this case you will need to rename the table manually. Following the upgrade, django CMS will look for the tables for these plugins under their new name, and will report that they don't exist if it can't find them.
- 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
```

3.0.16 release notes

Bug-fixes

- Fixed JavaScript error when using `PageSelectWidget`
- Fixed whitespace markup issues in draft mode
- Added plugin migrations layout detection in tests

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 pop-up dialogs
- Exclude `scssc` from manifest
- Remove unpublished pages from menu
- Remove page from menu items for performance reason
- Fix access to 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.

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 template tags
- Minor documentation fixes
- Fix an issue related to "Empty all" Placeholder feature
- Fix plugin sorting in Python 3
- 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 incompatible 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 behaviour 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.

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 reliability of Advanced Settings on page when using apphooks
- Allow page deletion after template removal
- Improves upstream caching accuracy
- Improves CMSAttachMenu registration
- Improves handling of mis-typed URLs
- Improves redirection as a result of changes to page slugs, etc.
- Improves performance of "watched models"
- Improves frontend performance relating to re-sizing the sideframe
- Corrects an issue where items might not be visible in structure 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.

3.0.12 release notes

What's new in 3.0.12

Bug Fixes

- Fixes a regression caused by extra whitespace in JavaScript

3.0.11 release notes

What's new in 3.0.11

- Core support for multiple instances of the same apphooked 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

3.0.10 release notes

What's new in 3.0.10

- Improved Python 3 compatibility
- Improved the behaviour 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

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

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 dependency to be PEP440 compatible
- Fix some Django 1.4 compatibility issues
- Add toolbar sanity check
- Fix behaviour with CMSPluginBase.get_render_template()
- Fix issue on django >= 1.6 with page form fields.
- Resolve jQuery namespace issues in admin page tree and change form
- Fix issues for PageField in Firefox/Safari
- Fix some Python 3.4 compatibility issue when using proxy modules
- Fix corner case in plugin copy
- Documentation fixes
- Minor code clean-ups

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.

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 documentation

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 *How to create 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*.

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 behaviour.

Docs on how to disable or customise this behaviour have a look here:

Apphook permissions

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: *See the quick upgrade guide*

New Frontend Editing

django CMS 3.0 introduces a new frontend editing system as well as a customisable 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 *How to extend the Toolbar*.

Warning: Upgrading from previous versions

3.0 now requires the `django.contrib.messages` application for the toolbar to work.

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 behaviour.

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/divio/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 any more. 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_templace.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 any more 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 pre-fillable with default plugins

In CMS_PLACEHOLDER_CONF, for each placeholder configuration, you can specify via 'default_plugins' a list of plugins to automatically 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.

Frontend 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 template tags 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 table-name magic for plugins

Since django CMS 2.0 plugins had their table names start with *cmsplugin_*. We removed this behaviour 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 schema migration and rename the table by hand.

> **Warning:** When working in the django shell or coding at low level, you **must** trigger the backward compatible behaviour (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 behaviour 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 following the "Explicit setup" instructions.

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-whatever`). 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 any more.

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)
```

Finally, your existing pages will be unpublished, so publish them with the `publisher` command:

```
python manage.py publisher_publish
```

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.

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 onward, 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 that 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 information 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 achieve the same functionality with django-i18nurl. It is a backport of the new functionality in django 1.4 and can be found here:

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

What you need to do:

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

```
from django.conf.urls 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 %}` template tag 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 behaviour 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'),
```

(continues on next page)

(continued from previous page)

```
                '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 template tag 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 behaviour with the new *CMS_MAX_PAGE_PUBLISH_REVERSIONS* setting.

Changes to the show_sub_menu template tag

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 `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 behaviour 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 bug-fix makes this redundant as every non-managed URL will behave like this.

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 time zones

On Django 1.4, and with `USE_TZ=True` the django CMS now uses time zone 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-populate the slug field if `DEBUG` was set to `False`. Again, this issue is now resolved.

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.

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 data migration has been introduced to modify existing Googlemap plugin instance to set the default value if *level* if is *NULL*.

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.
- Internationalised 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 any more:

```
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.

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 favour 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.plugin_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 favour of the built-in multi-lingual support mechanisms.

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` sub-folder exists in your project and is writeable.

Note: PROJECT_PATH is the absolute path to your project.

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>
```

(continues on next page)

(continued from previous page)

```
        {% 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
```

4.1.7 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.

If you don't have an IRC client, you can join our IRC channel using the KiwiIRC web client, which works pretty well.

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 custom changes 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.

Tutorial

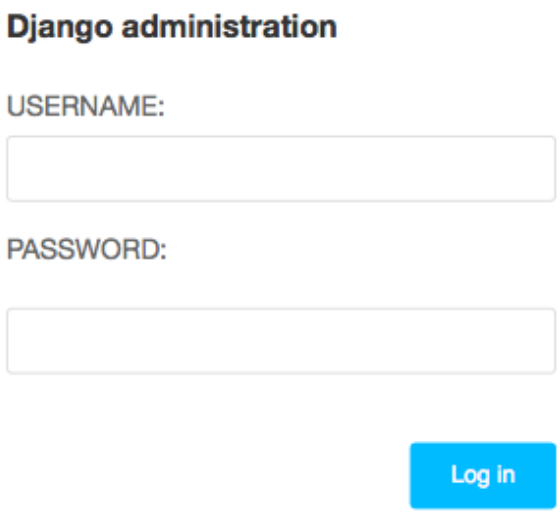
Note: This is a new section in the django CMS documentation, and a priority for the project. If you'd like to contribute to it, we'd love to hear from you - join us on the #django-cms IRC channel on freenode or the django-cms-developers email list.

If you don't have an IRC client, you can join our IRC channel using the KiwiIRC web client, which works pretty well.

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

When you visit a brand new site for the first time, you will be invited to log in.

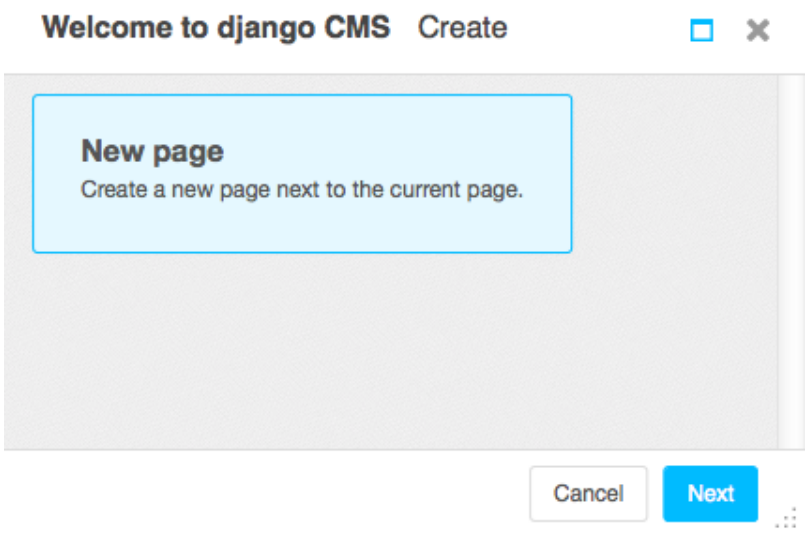


The developers of your site are responsible for creating and providing the login credentials so consult them if you are unsure.

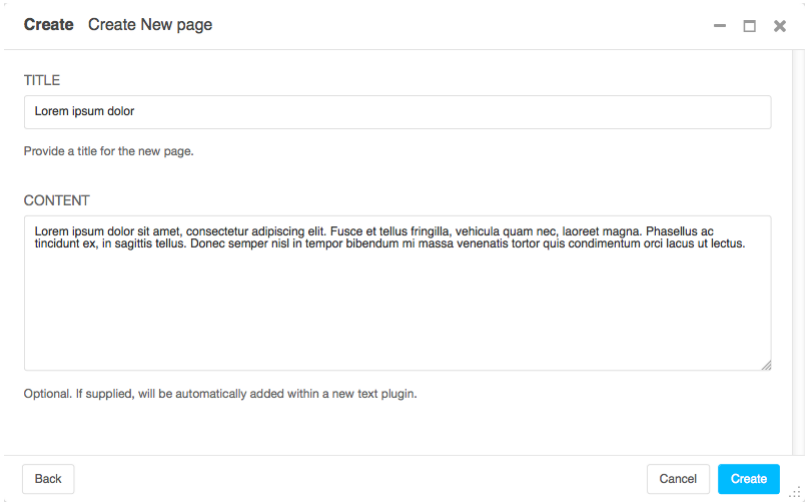
Create a page

Create your first page

django CMS's *Create Page wizard* will open a new dialog box.

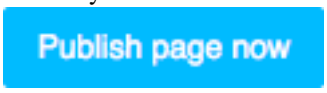


Select *Next*, and provide a *Title* and some basic text content for the new page (you'll be able to add formatting to this text in a moment), then hit **Create**.

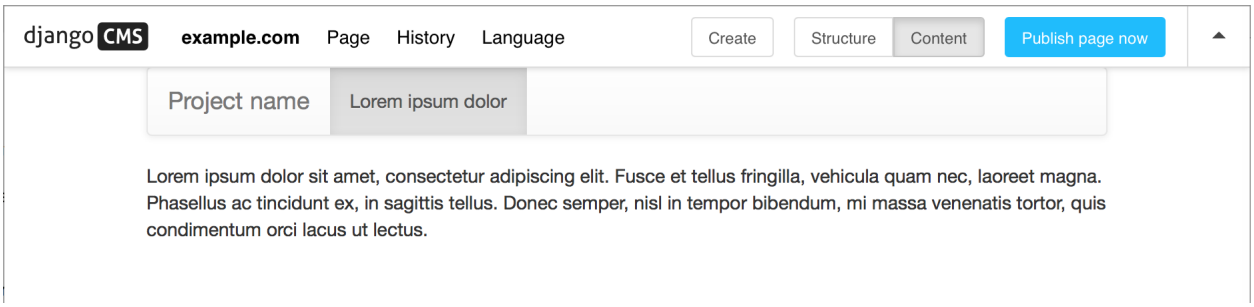


Here's your newly-created page, together with the *django CMS toolbar*, your primary tool for managing django CMS content.

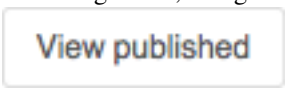
Publish a page

Your newly-created page is just a *draft*, and won't actually be published until you decide. As an editor, you can see drafts, but other visitors to your site will only see published pages. Hit 

to publish it.



To edit the page further, switch back into editing mode, using the 

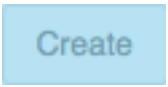
button that appears, and return to the *published* version of the page using the 

button.

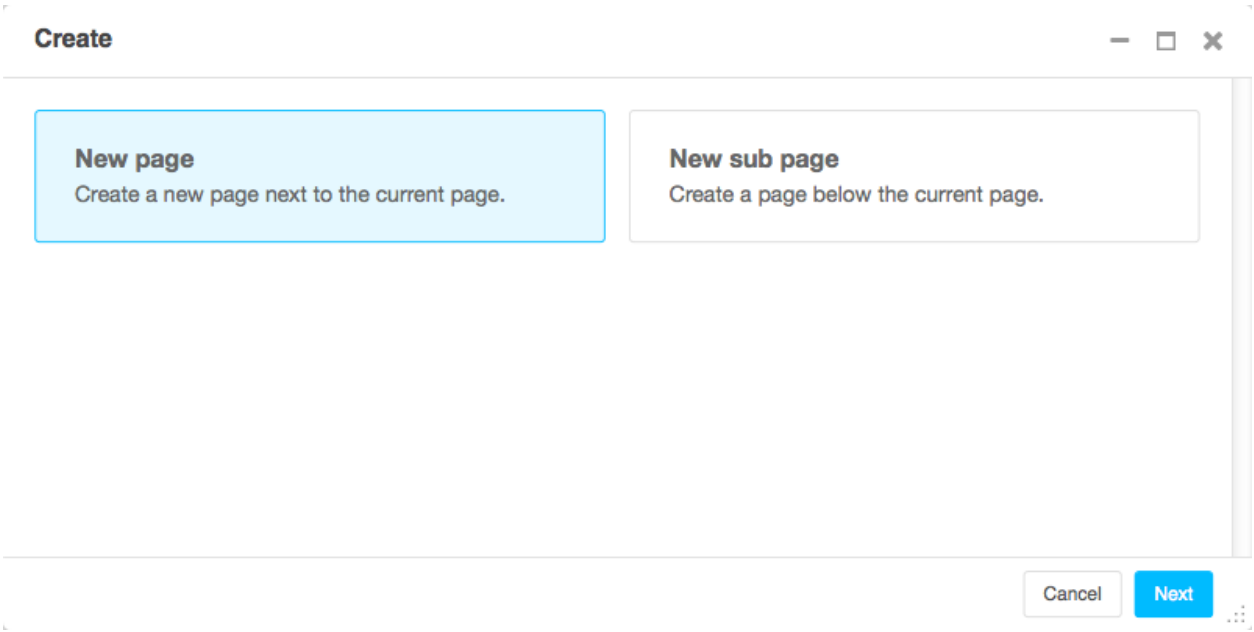
In editing mode, double-click on the paragraph of text to change it. This will open the Text plugin containing it. Make changes, add some formatting, and **Save** it again.

You can continue making and previewing changes privately until you are ready to publish them.

Create a second page

Hit 

to create a second page. This opens the *Create page* dialog:



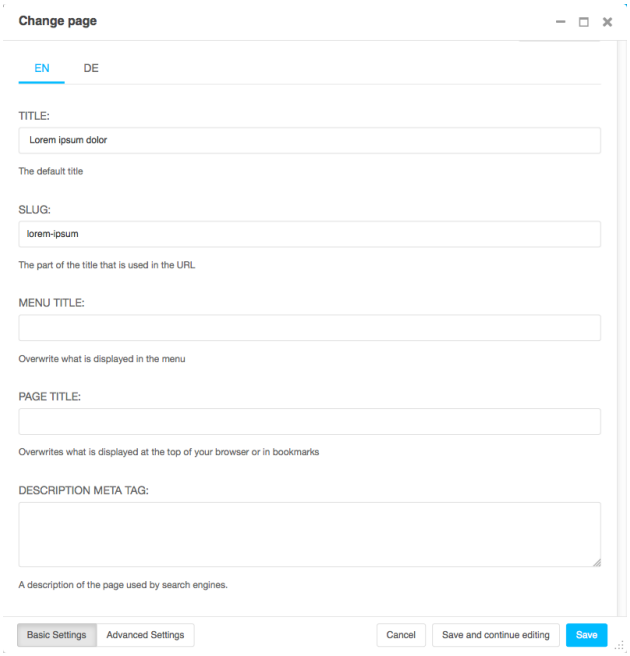
In django CMS, pages can be arranged hierarchically. This is important for larger sites. Choose whether the new page should be a sub-page - a child - of the existing page, or be on the same level in the hierarchy - a sibling.

Once again, give the page a *Title* and some basic text content. Continue making changes to content and formatting, and then **Publish** it as you did previously.

Changing page settings

The django CMS toolbar offers other useful editing tools.

Switch to *Edit* mode on one of your pages, and from the toolbar select *Page* > *Page settings...*. The *Change page* dialog that opens allows you to manage key settings for your page.

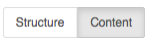


Some key settings:

- *Slug*: The page's *slug* is used to form its URL. For example, a page *Lenses* that is a sub-page of *Photography* might have a URL that ends `photography/lenses`. You can change the automatically-generated slug of a page if you wish to. Keep slugs short and meaningful, as they are useful to human beings and search engines alike.
- *Menu Title*: If you have a page called *Photography: theory and practice*, you might not want the whole title to appear in menus - shortening it to *Photography* would make more sense.
- *Page Title*: By default, a page's `<title>` element is taken from the *Title*, but you can override this here. The `<title>` element isn't displayed on the page, but is used by search engines and web browsers - as far as they are concerned, it's the page's real title.
- *Description meta tag*: A short piece of text that will be used by search engines (and displayed in lists of search results) and other indexing systems.

There are also some *Advanced Settings*, but you don't need to be concerned about these now.

Structure and content modes

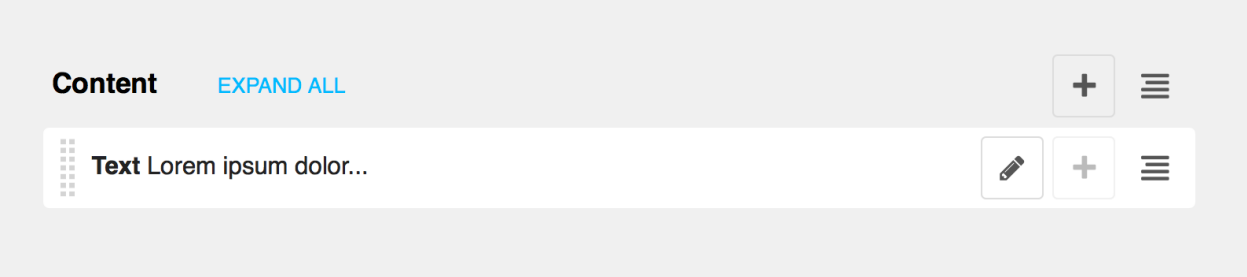


The *Structure/Content* mode control in the toolbar lets you switch between two different editing modes.

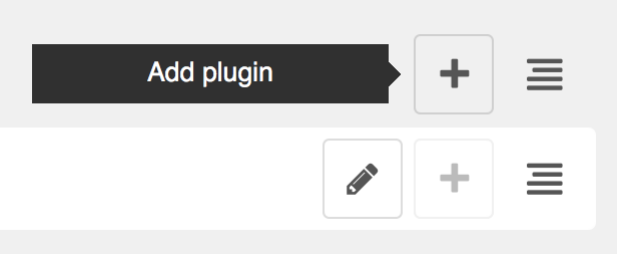
You've already used *Content* mode, in which you can double-click on content to edit it.

In *Structure* mode, you can manage the placement of content within the page structure.

Switch to *Structure* mode. This reveals the *structure board* containing the *placeholders* available on the page, and the *plugins* in them:



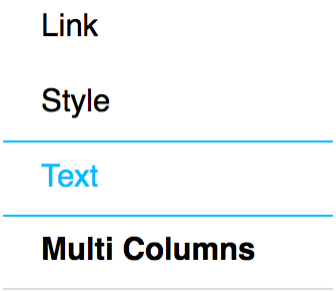
Here there is just one placeholder, called *Content*, containing one plugin - a text plugin that begins *Lorem ipsum dolor...*.



Add a second plugin

Let's add another plugin.

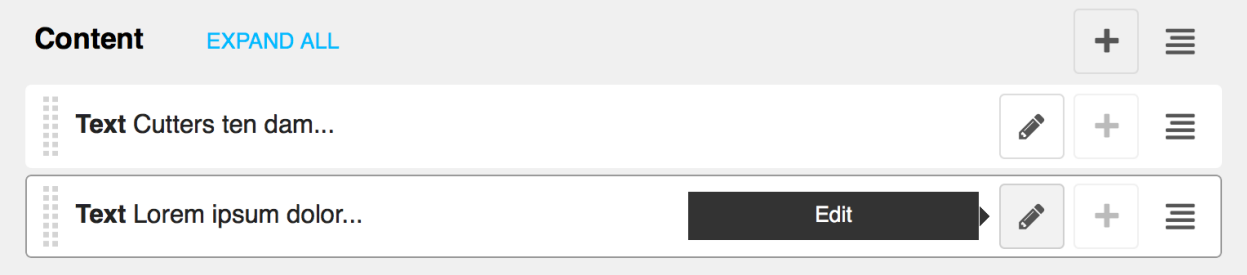
Select the **Add plugin** icon (+) and choose *Text* from the list of available plugin types.



This will open a familiar text editor; add some text and **Save**. Now in the structure board you'll see the new *Text* plugin - which you can move around within the structure, to re-order the plugins.

Note: You don't need to save these changes manually - they are saved automatically as soon as you make them. However, they still need to be published in order for other users to see them.

Each plugin in the structure board is available for editing by double-clicking or selecting the edit icon.



You can switch back to content mode to see the effect of your changes, and **Publish** the page to make them public.

Note: Touch-screen users

django CMS supports touch-screen interfaces, though there are currently some limitations in support. You will be able to complete the tutorial using a touch-screen device, but please consult *Using touch-screen devices with django CMS*, and see the notes on *Device support*.

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.

If you don't have an IRC client, you can join our IRC channel using the KiwiIRC web client, which works pretty well.

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

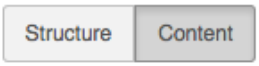
Language allows you to switch to a different language version of the page you're on, and manage the various translations.

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.

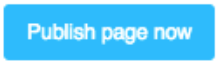
The *Structure/Content* button

Allows you to switch between different editing modes (when you're looking at a draft only).



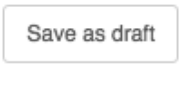
Publishing controller

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

- **View published** does the same as "Save as draft"

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:

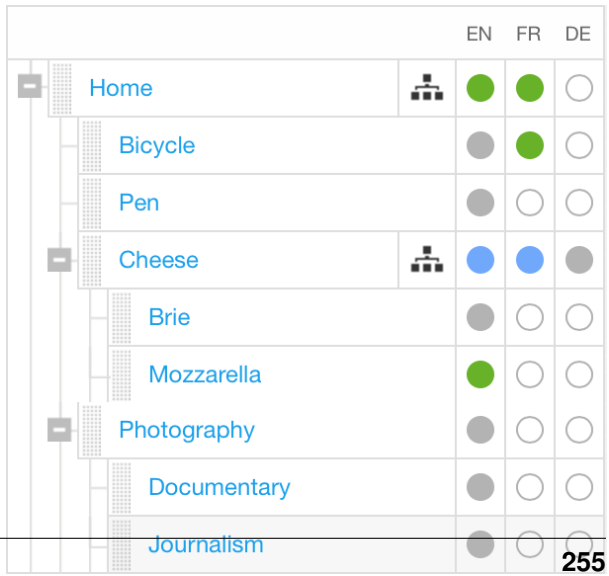
The page you're currently on is highlighted in grey (in this case, *Journalism*, the last in the list).

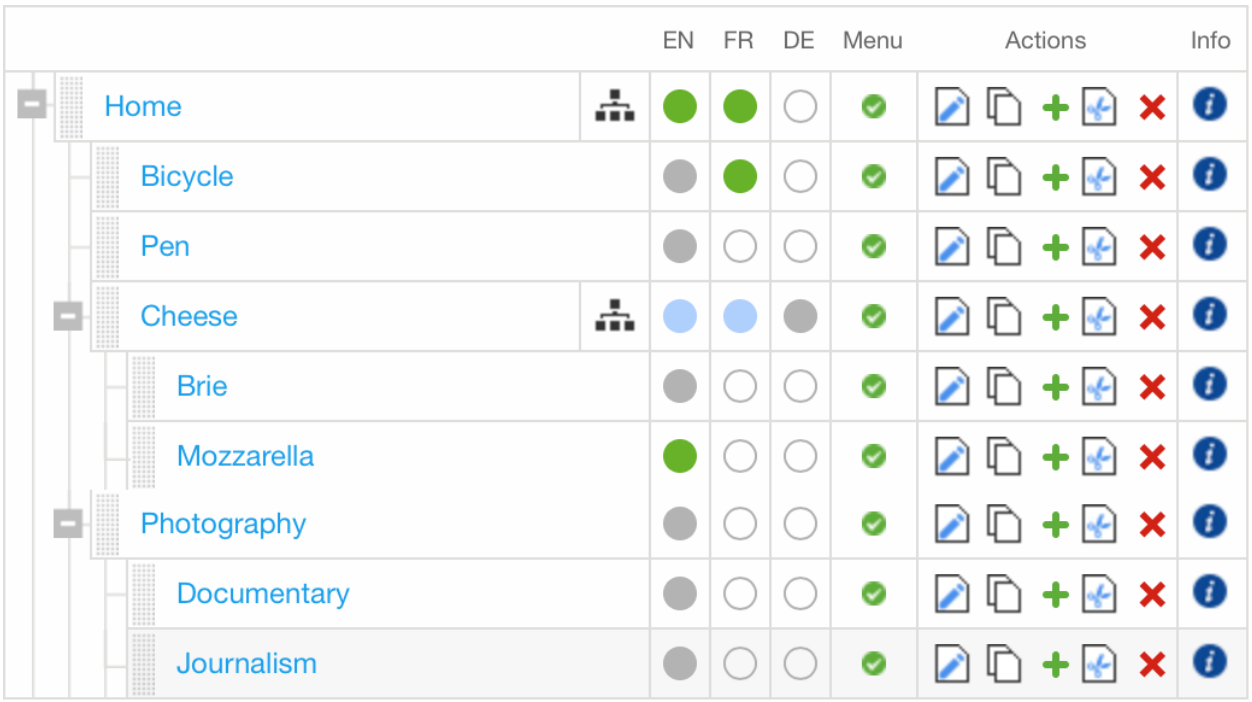
From left to right, items in the list have:

- 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





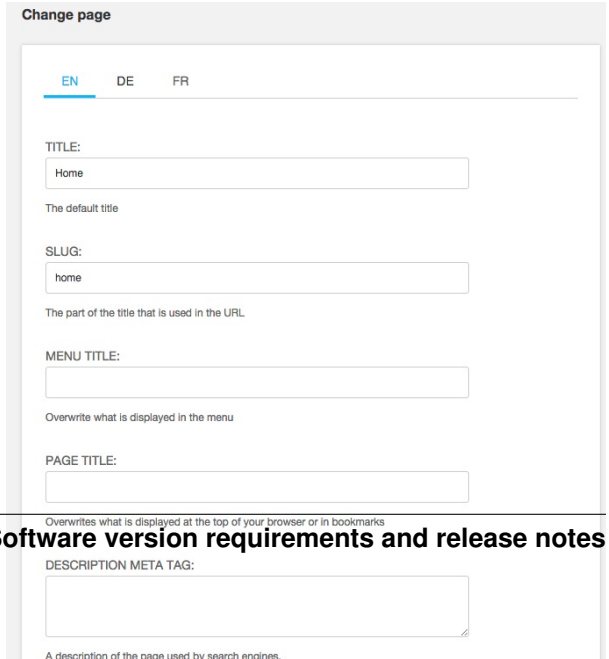
Basic page settings

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 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*.

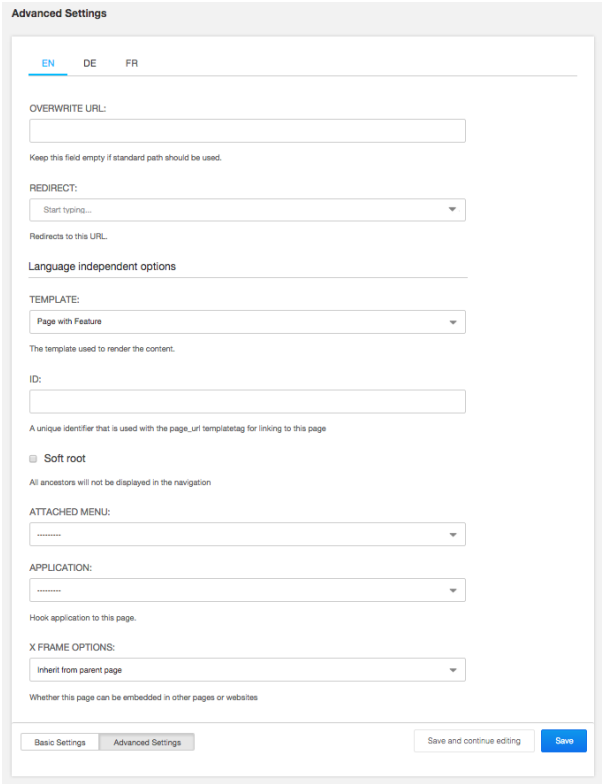
Page title is expected to be used by django CMS templates for the <*title*> element of the page (which will otherwise simply use the *Title* field). If provided, it will be the *Page title* that appears in the browser's title bar and bookmarks, and in search engine results.

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 typically 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 it 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
    - 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.

Python Module Index

C

cms.api, 113
cms.app_base, 117
cms.constants, 117
cms.extensions.toolbar, 166
cms.management, 118
cms.models, 142
cms.templatetags.cms_tags, 150
cms.toolbar.items, 164
cms.toolbar.toolbar, 163
cms.toolbar_pool, 166

Index

Symbols

__init__() (*cms.plugin_base.PluginMenuItem method*), 148
_build_nodes() (*menus.menu_pool.MenuPool method*), 141
_mark_selected() (*menus.menu_pool.MenuPool method*), 141
_menus (*cms.app_base.CMSApp attribute*), 117
_setup_extension_toolbar() (*cms.extensions.toolbar.ExtensionToolbar method*), 166
_urls (*cms.app_base.CMSApp attribute*), 117

A

accepted, **185**
ACCESS_CHILDREN (*in module cms.models*), 142
ACCESS_DESCENDANTS (*in module cms.models*), 142
ACCESS_PAGE (*in module cms.models*), 142
ACCESS_PAGE_AND_DESCENDANTS (*in module cms.models*), 142
add_ajax_item() (*cms.toolbar.items.ToolbarMixin method*), 165
add_break() (*cms.toolbar.items.Menu method*), 166
add_button() (*cms.toolbar.items.ButtonList method*), 166
add_button() (*cms.toolbar.toolbar.CMSToolbar method*), 164
add_button_list() (*cms.toolbar.toolbar.CMSToolbar method*), 164
add_item() (*cms.toolbar.items.ButtonList method*), 166
add_item() (*cms.toolbar.items.ToolbarMixin method*), 164
add_item() (*cms.toolbar.toolbar.CMSToolbar method*), 163
add_link_item() (*cms.toolbar.items.ToolbarMixin method*), 165
add_modal_item() (*cms.toolbar.items.ToolbarMixin method*), 165
add_plugin() (*in module cms.api*), 115
add_sideframe_item() (*cms.toolbar.items.ToolbarMixin method*), 165
admin_preview (*cms.plugin_base.CMSPluginBase attribute*), 143
AjaxItem (*class in cms.toolbar.items*), 166
allow_children (*cms.plugin_base.CMSPluginBase attribute*), 143
apply_modifiers() (*menus.menu_pool.MenuPool method*), 141
assign_user_to_page() (*in module cms.api*), 115
attr (*menus.base.NavigationNode attribute*), 141, 142
AUTH_USER_MODEL
    setting, 122

B

backport, **187**
BaseItem (*class in cms.toolbar.items*), 165
blocker, **187**
Break (*class in cms.toolbar.items*), 166
build_mode (*cms.toolbar.toolbar.CMSToolbar attribute*), 163
Button (*class in cms.toolbar.items*), 166
ButtonList (*class in cms.toolbar.items*), 166

C

cache (*cms.plugin_base.CMSPluginBase attribute*), 143
change_form_template (*cms.plugin_base.CMSPluginBase attribute*), 143
child_classes (*cms.plugin_base.CMSPluginBase attribute*), 144
cms.admin.placeholderadmin.PlaceholderAdminMixin (*built-in class*), 143
cms.api (*module*), 113
cms.app_base (*module*), 117
cms.constants (*module*), 117
cms.extensions.toolbar (*module*), 166

cms.forms.fields.PageSelectFormField (*built-in class*), 136
cms.forms.fields.PageSmartLinkField (*built-in class*), 136
cms.management (*module*), 118
cms.menu.CMSMenu (*built-in class*), 142
cms.menu.NavExtender (*built-in class*), 142
cms.menu.SoftRootCutter (*built-in class*), 142
cms.menu_bases.CMSAttachMenu (*built-in class*), 142
cms.models (*module*), 142
cms.models.fields.PageField (*built-in class*), 136
cms.models.fields.PlaceholderField (*built-in class*), 136
cms.models.Page (*built-in class*), 142
cms.models.placeholdermodel.Placeholder (*built-in class*), 143
cms.models.pluginmodel.CMSPlugin (*built-in class*), 148
cms.models.Title (*built-in class*), 163
cms.plugin_base.CMSPluginBase (*built-in class*), 143
cms.plugin_base.PluginMenuItem (*built-in class*), 148
cms.plugin_pool.PluginPool (*built-in class*), 150
cms.sitemaps.CMSSitemap (*built-in class*), 150
cms.templatetags.cms_tags (*module*), 150
cms.toolbar.items (*module*), 164
cms.toolbar.toolbar (*module*), 163
cms.toolbar_pool (*module*), 166
CMS_APPHOOKS
    setting, 127
CMS_CACHE_DURATIONS
    setting, 133
CMS_CACHE_PREFIX
    setting, 133
CMS_INTERNAL_IPS
    setting, 132
CMS_LANGUAGES
    setting, 127
CMS_MAX_PAGE_PUBLISH_REVERSIONS
    setting, 134
CMS_MEDIA_PATH
    setting, 131
CMS_MEDIA_ROOT
    setting, 131
CMS_MEDIA_URL
    setting, 131
CMS_PAGE_CACHE
    setting, 134
CMS_PAGE_MEDIA_PATH
    setting, 131
CMS_PAGE_WIZARD_CONTENT_PLACEHOLDER
    setting, 136
CMS_PAGE_WIZARD_CONTENT_PLUGIN
    setting, 136
CMS_PAGE_WIZARD_CONTENT_PLUGIN_BODY
    setting, 136
CMS_PAGE_WIZARD_DEFAULT_TEMPLATE
    setting, 135
CMS_PERMISSION
    setting, 132
CMS_PLACEHOLDER_CACHE
    setting, 134
CMS_PLACEHOLDER_CONF
    setting, 124
CMS_PLUGIN_CACHE
    setting, 134
CMS_PLUGIN_CONTEXT_PROCESSORS
    setting, 127
CMS_PLUGIN_PROCESSORS
    setting, 127
CMS_PUBLIC_FOR
    setting, 133
CMS_RAW_ID_USERS
    setting, 132
CMS_REQUEST_IP_RESOLVER
    setting, 132
CMS_TEMPLATE_INHERITANCE
    setting, 123
CMS_TEMPLATES
    setting, 122
CMS_TEMPLATES_DIR
    setting, 123
cms_toolbar
    template tag, 162
CMS_TOOLBARS
    setting, 134
CMS_UNIHANDECODE_DECODERS
    setting, 130
CMS_UNIHANDECODE_DEFAULT_DECODER
    setting, 130
CMS_UNIHANDECODE_HOST
    setting, 130
CMS_UNIHANDECODE_VERSION
    setting, 130
CMSApp (*class in cms.app_base*), 117
CMSToolbar (*class in cms.toolbar.toolbar*), 163
code
    setting, 128
copy_relations() (*cms.models.pluginmodel.CMSPlugin method*), 148
create_page() (*in module cms.api*), 114
create_page_user() (*in module cms.api*), 115
create_title() (*in module cms.api*), 114

csrf_token (*cms.toolbar.toolbar.CMSToolbar attribute*), 163

D

design decision, **186**
disable_child_plugins (*cms.plugin_base.CMSPluginBase attribute*), 144
discover_menus() (*menus.menu_pool.MenuPool method*), 141
docs, **186**

E

easy pickings, **187**
edit_mode (*cms.toolbar.toolbar.CMSToolbar attribute*), 163
expert opinion, **186**
EXPIRE_NOW (*in module cms.constants*), 117
ExtensionToolbar (*class in cms.extensions.toolbar*), 166

F

fallbacks
    setting, 129
find_first() (*cms.toolbar.items.ToolbarMixin method*), 165
find_items() (*cms.toolbar.items.ToolbarMixin method*), 165
form (*cms.plugin_base.CMSPluginBase attribute*), 144
frontend_edit_template (*cms.plugin_base.CMSPluginBase attribute*), 144

G

get_absolute_url() (*menus.base.NavigationNode method*), 142
get_add_url() (*cms.models.pluginmodel.CMSPlugin method*), 149
get_alphabetical_insert_position() (*cms.toolbar.items.ToolbarMixin method*), 164
get_ancestors() (*menus.base.NavigationNode method*), 141
get_cache_expiration() (*cms.plugin_base.CMSPluginBase method*), 146
get_config() (*cms.app_base.CMSApp method*), 117
get_config_add_url() (*cms.app_base.CMSApp method*), 117
get_configs() (*cms.app_base.CMSApp method*), 117
get_context() (*cms.toolbar.items.BaseItem method*), 165
get_context() (*menus.templatetags.menu_tags.ShowMenu method*), 141
get_copy_url() (*cms.models.pluginmodel.CMSPlugin method*), 149
get_delete_url() (*cms.models.pluginmodel.CMSPlugin method*), 149
get_descendants() (*menus.base.NavigationNode method*), 141
get_edit_url() (*cms.models.pluginmodel.CMSPlugin method*), 149
get_extra_global_plugin_menu_items() (*cms.plugin_base.CMSPluginBase method*), 146
get_extra_local_plugin_menu_items() (*cms.plugin_base.CMSPluginBase method*), 146
get_extra_placeholder_menu_items() (*cms.plugin_base.CMSPluginBase method*), 145
get_item_count() (*cms.toolbar.items.ToolbarMixin method*), 164
get_menu() (*cms.toolbar.toolbar.CMSToolbar method*), 164
get_menu_title() (*menus.base.NavigationNode method*), 142
get_menus() (*cms.app_base.CMSApp method*), 117
get_move_url() (*cms.models.pluginmodel.CMSPlugin method*), 149
get_nodes() (*menus.base.Menu method*), 140
get_nodes() (*menus.menu_pool.MenuPool method*), 141
get_or_create_menu() (*cms.toolbar.items.Menu method*), 166
get_or_create_menu() (*cms.toolbar.toolbar.CMSToolbar method*), 164
get_page_extension_admin() (*cms.extensions.toolbar.ExtensionToolbar method*), 166
get_plugin_urls() (*cms.plugin_base.CMSPluginBase method*), 145
get_render_template() (*cms.plugin_base.CMSPluginBase method*), 145
get_title_extension_admin() (*cms.extensions.toolbar.ExtensionToolbar method*), 166
get_translatable_content() (*cms.models.pluginmodel.CMSPlugin method*), 149
get_urls() (*cms.app_base.CMSApp method*), 118
get_vary_cache_on() (*cms.plugin_base.CMSPluginBase method*), 146

H

has patch, **187**
hide_untranslated
    setting, 129

I

icon_alt() (*cms.plugin_base.CMSPluginBase method*), 146
icon_src() (*cms.plugin_base.CMSPluginBase method*), 147
index (*cms.toolbar.items.ItemSearchResult attribute*), 164
is_staff (*cms.toolbar.toolbar.CMSToolbar attribute*), 163
item (*cms.toolbar.items.ItemSearchResult attribute*), 164
ItemSearchResult (*class in cms.toolbar.items*), 164

L

language_chooser
    template tag, 162
LEFT (*cms.toolbar.items.ToolbarMixin attribute*), 164
LEFT (*in module cms.constants*), 117
LinkItem (*class in cms.toolbar.items*), 166

M

mark_levels() (*menus.modifiers.Level method*), 142
marked for rejection, **186**
MAX_EXPIRATION_TTL (*in module cms.constants*), 117
Menu (*class in cms.toolbar.items*), 165
menus.base.Menu (*built-in class*), 140
menus.base.Modifier (*built-in class*), 141
menus.base.NavigationNode (*built-in class*), 141
menus.menu_pool._build_nodes_inner_for_one_menu (*built-in function*), 141
menus.menu_pool.MenuPool (*built-in class*), 141
menus.modifiers.AuthVisibility (*built-in class*), 142
menus.modifiers.Level (*built-in class*), 142
menus.modifiers.Marker (*built-in class*), 142
menus.templatetags.menu_tags.cut_levels() (*built-in function*), 141
menus.templatetags.menu_tags.ShowMenu (*built-in class*), 141
ModalItem (*class in cms.toolbar.items*), 166
model (*cms.plugin_base.CMSPluginBase attribute*), 144
modify() (*menus.base.Modifier method*), 141
module (*cms.plugin_base.CMSPluginBase attribute*), 144
more info, **186**

N

name (*cms.plugin_base.CMSPluginBase attribute*), 144
non-issue, **185**

O

on hold, **187**

P

page_attribute
    template tag, 154
page_language_url
    template tag, 161
page_lookup
    template tag, 153
page_only (*cms.plugin_base.CMSPluginBase attribute*), 144
page_url
    template tag, 154
PagePermission (*class in cms.models*), 142
parent_classes (*cms.plugin_base.CMSPluginBase attribute*), 144
patch, **186**
placeholder
    template tag, 150
post_copy() (*cms.models.pluginmodel.CMSPlugin method*), 149
public
    setting, 129
publish_page() (*in module cms.api*), 115
publish_pages() (*in module cms.api*), 116

R

ready for review, **186**
ready to be merged, **186**
redirect_on_fallback
    setting, 129
REFRESH (*in module cms.constants*), 117
REFRESH_PAGE (*cms.toolbar.items.ToolbarMixin attribute*), 164
register() (*cms.toolbar_pool.ToolbarPool method*), 166
remove_item() (*cms.toolbar.items.ToolbarMixin method*), 165
remove_item() (*cms.toolbar.toolbar.CMSToolbar method*), 163
render() (*cms.plugin_base.CMSPluginBase method*), 147
render() (*cms.toolbar.items.BaseItem method*), 165
render_model
    template tag, 156
render_model_add
    template tag, 160
render_model_add_block
    template tag, 161
render_model_block
    template tag, 157

render_model_icon
    template tag, 159
render_placeholder
    template tag, 152
render_plugin
    template tag, 155
render_plugin (*cms.plugin_base.CMSPluginBase attribute*), 145
render_plugin_block
    template tag, 155
render_template (*cms.plugin_base.CMSPluginBase attribute*), 145
render_uncached_placeholder
    template tag, 152
require_parent (*cms.plugin_base.CMSPluginBase attribute*), 145
RIGHT (*cms.toolbar.items.ToolbarMixin attribute*), 164
RIGHT (*in module cms.constants*), 117

S

set_translatable_content() (*cms.models.pluginmodel.CMSPlugin method*), 149
setting
    AUTH_USER_MODEL, 122
    CMS_APPHOOKS, 127
    CMS_CACHE_DURATIONS, 133
    CMS_CACHE_PREFIX, 133
    CMS_INTERNAL_IPS, 132
    CMS_LANGUAGES, 127
    CMS_MAX_PAGE_PUBLISH_REVERSIONS, 134
    CMS_MEDIA_PATH, 131
    CMS_MEDIA_ROOT, 131
    CMS_MEDIA_URL, 131
    CMS_PAGE_CACHE, 134
    CMS_PAGE_MEDIA_PATH, 131
    CMS_PAGE_WIZARD_CONTENT_PLACEHOLDER, 136
    CMS_PAGE_WIZARD_CONTENT_PLUGIN, 136
    CMS_PAGE_WIZARD_CONTENT_PLUGIN_BODY, 136
    CMS_PAGE_WIZARD_DEFAULT_TEMPLATE, 135
    CMS_PERMISSION, 132
    CMS_PLACEHOLDER_CACHE, 134
    CMS_PLACEHOLDER_CONF, 124
    CMS_PLUGIN_CACHE, 134
    CMS_PLUGIN_CONTEXT_PROCESSORS, 127
    CMS_PLUGIN_PROCESSORS, 127
    CMS_PUBLIC_FOR, 133
    CMS_RAW_ID_USERS, 132
    CMS_REQUEST_IP_RESOLVER, 132
    CMS_TEMPLATE_INHERITANCE, 123
    CMS_TEMPLATES, 122
    CMS_TEMPLATES_DIR, 123
    CMS_TOOLBARS, 134
    CMS_UNIHANDECODE_DECODERS, 130
    CMS_UNIHANDECODE_DEFAULT_DECODER, 130
    CMS_UNIHANDECODE_HOST, 130
    CMS_UNIHANDECODE_VERSION, 130
    code, 128
    fallbacks, 129
    hide_untranslated, 129
    public, 129
    redirect_on_fallback, 129
show_breadcrumb
    template tag, 139
show_menu
    template tag, 137
show_menu_below_id
    template tag, 138
show_placeholder
    template tag, 152
show_sub_menu
    template tag, 138
show_toolbar (*cms.toolbar.toolbar.CMSToolbar attribute*), 163
show_uncached_placeholder
    template tag, 153
side (*cms.toolbar.items.BaseItem attribute*), 165
SideframeItem (*class in cms.toolbar.items*), 166
static_placeholder
    template tag, 151
SubMenu (*class in cms.toolbar.items*), 166

T

template (*cms.toolbar.items.BaseItem attribute*), 165
template tag
    cms_toolbar, 162
    language_chooser, 162
    page_attribute, 154
    page_language_url, 161
    page_lookup, 153
    page_url, 154
    placeholder, 150
    render_model, 156
    render_model_add, 160
    render_model_add_block, 161
    render_model_block, 157
    render_model_icon, 159
    render_placeholder, 152
    render_plugin, 155
    render_plugin_block, 155
    render_uncached_placeholder, 152
    show_breadcrumb, 139
    show_menu, 137
    show_menu_below_id, 138

show_placeholder, 152
show_sub_menu, 138
show_uncached_placeholder, 153
static_placeholder, 151
TEMPLATE_INHERITANCE_MAGIC (*in module cms.constants*), 117
tests, **186**
text_editor_button_icon() (*cms.plugin_base.CMSPluginBase method*), 148
text_enabled (*cms.plugin_base.CMSPluginBase attribute*), 145
toolbar_language (*cms.toolbar.toolbar.CMSToolbar attribute*), 163
ToolbarMixin (*class in cms.toolbar.items*), 164
ToolbarPool (*class in cms.toolbar_pool*), 166
translatable_content_excluded_fields (*cms.models.pluginmodel.CMSPlugin attribute*), 148

V

VISIBILITY_ALL (*in module cms.api*), 113
VISIBILITY_ANONYMOUS (*in module cms.api*), 114
VISIBILITY_USERS (*in module cms.api*), 113

W

watch_models (*cms.toolbar.toolbar.CMSToolbar attribute*), 163
won't fix, **186**
work in progress, **186**