django cms Documentation

Release 5.1.0a1

django CMS Association and contributors

Jul 03, 2026

CONTENTS

Welcome to the **django CMS** developer documentation.

django CMS is a modern, open-source content management system built on Django, designed to solve complex publishing requirements using simple, composable parts.

This documentation is organised using the **Diátaxis framework**, which separates learning material, practical guides, conceptual explanations, and technical reference — so you can quickly find the information you need.

If you are looking for editor or administrator documentation, see the django CMS User Guide.

CHAPTER

ONE

IMMEDIATE PET PROJECT

Create a fully configured django CMS project in minutes:

```
python -m venv .venv
source .venv/bin/activate  # On Windows: .venv\Scripts\activate
pip install django-cms
djangocms mysite
```

Already have a Django project? Run `djangocms .` in its root directory to *add django CMS to it*. See the *djangocms command* reference for all options.

CHAPTER

TWO

CHOOSE YOUR PATH

Tutorials Step-by-step lessons that teach django CMS from installation through building your first project.

Tutorial How-to guides Practical, goal-oriented guides for solving specific problems in real projects.

How-to guides Explanation Background, concepts, and architectural decisions explained to help you understand how django CMS works.

Explanation Reference Authoritative technical reference for APIs, settings, commands, and internals.

Reference

CHAPTER

THREE

WHY DJANGO CMS?

django CMS is a mature, open-source content management system built on Django. It is designed for projects that require flexibility, long-term stability, and close integration with custom Django applications.

Key features include:

- robust internationalisation (i18n) and multi-site support
- front-end (and inline) editing that allows editors to work directly on rendered pages
- a flexible placeholder and plugin system for composing reusable content components
- integration with multiple rich-text editors
- support for content versioning, editorial workflows, and headless setups through official add-on packages

Compared to other Django-based CMS platforms, django CMS stands out for:

- a small, stable core that integrates cleanly into existing Django projects
- non-monolithic architecture that allows incremental adoption
- thorough, structured documentation organised using the Diátaxis framework
- an active, long-running open-source community
- an emphasis on code quality, testing, and long-term support

CHAPTER

FOUR

COMMUNITY & CONTRIBUTION

django CMS is developed and maintained by an open community. Participation is welcome at every level, from asking questions to improving documentation or contributing code.

Community chat Join the Discord server to ask questions and talk with other django CMS users and contributors.

https://discord-support-channel.django-cms.org Improve the docs Documentation improvements — including small fixes — are one of the easiest ways to contribute.

Contribute About the project Learn more about the people and organisations behind django CMS.

Who is behind django CMS

CHAPTER

FIVE

VERSIONS, COMPATIBILITY & SUPPORT

This documentation refers to django CMS version 5.1.0a1.

django CMS follows Django's long-term support (LTS) policy and provides aligned LTS releases.

5.1 Current LTS versions

django CMS	Feature freeze	Django	End of long-term support
x.x.x	February 2027	6.2	March 2030
5.0.x	February 2025	5.2	March 2028

After feature freeze, new features are developed for the next major django CMS release.

5.2 Unsupported LTS versions

The following LTS versions are **no longer supported**:

django CMS	Feature freeze	Django	End of long-term support
4.1.x	September 2023	4.2	March 2026
3.11.x	September 2023	3.2	March 2024
3.11.x	September 2023	4.2	March 2026
3.8.x	October 2020	2.2	March 2022
3.7.x	June 2020	2.2	March 2022

5.3 Django / Python compatibility

LTS indicates a Django and django CMS combination covered by long-term support.

✓ means tested and supported. × means untested or incompatible.

Django CMS	Python						Django					
	3.14	3.13	3.12	3.11	3.10	3.9	6.1	6.0	5.2	5.1	5.0	4.2
5.1.x	✓	✓	✓	✓	✓	×	✓	✓	✓	×	×	×
5.0.x	✓	✓	✓	✓	✓	✓	×	✓	LTS	✓	✓	✓
4.1.x	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	LTS
3.11.x	×	×	✓	✓	✓	✓	×	×	×	✓	✓	LTS

For dependency details, see the project's `pyproject.toml` or the *Release notes & upgrade information*.

The *Commonly Used Plugin section* lists additional packages commonly used in django CMS projects.

5.3.1 Tutorial

A guided walkthrough that builds a small, real-looking django CMS site from an empty project. You will end up with a marketing site for a fictional coffee roaster — a few editable pages, a custom plugin, and a catalogue mounted via an apphook.

The tutorial is structured for someone who:

- has used Django at the level of the official Django tutorial,
- has never (or barely) used django CMS,
- wants to feel productive within an hour.

Plan to spend roughly **30 minutes reading and 60 minutes typing** for the whole sequence. Each chapter ends with something you can see in the browser.

Conventions

- The *Explanation* section answers *why*. This tutorial does not.
- The *Reference* section is the authoritative API. This tutorial cites it but never reproduces it.
- The *How-to guides* section covers production concerns (caching, multi-language, deployment, headless). This tutorial does not.

When something in the tutorial feels too small, that is on purpose. The goal is to finish a working site, not to enumerate every option.

The chapters

Installing django CMS

This is a short setup chapter. Its only goal is to get you to a running `python -m manage runserver` with the CMS welcome screen visible at `http://localhost:8000/`. Everything after this is the real tutorial.

Prerequisites

You need:

- **Python** 3.10 and `pip` working from your shell.
- **Django** familiarity at the level of the Django tutorial — settings files, `manage.py`, apps, templates.
- A text editor and a terminal.

You do **not** need: prior django CMS experience, a database server, or any frontend tooling. The tutorial uses SQLite.

Install django CMS

Open a terminal and run:

```
python3 -m venv .venv
source .venv/bin/activate  # On Windows: .venv\Scripts\activate
pip install django-cms
djangocms coffeesite
```

The `djangocms` command is a shortcut that creates a fully wired django CMS project. Behind the scenes it:

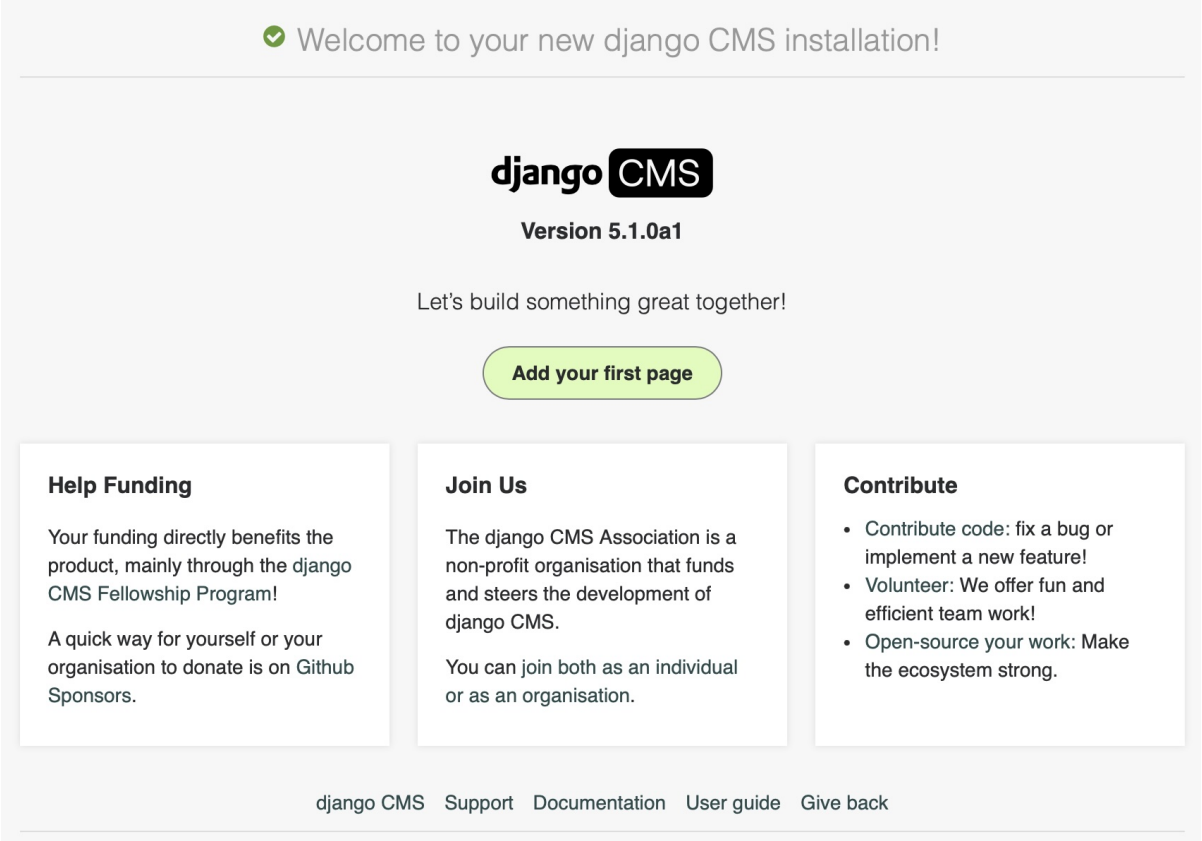
1. Runs `django-admin startproject coffeesite` with the official cms-template project template.
2. Installs the packages the template depends on: djangocms-text (rich-text editor), djangocms-frontend (Bootstrap 5 support), django-filer (media files), djangocms-versioning (draft / published versions), djangocms-alias (reusable content blocks), djangocms-simple-admin-style (admin theming).
3. Runs `python -m manage migrate` to create the SQLite database.
4. Prompts you to create a superuser.
5. Runs `python -m manage cms check` to verify the install.

The tutorial uses the project name `coffeesite`. If you pick a different name, substitute it everywhere you see `coffeesite` below.

Run the development server

```
cd coffeesite
python -m manage runserver
```

Open `http://localhost:8000/` in your browser. You should see the django CMS welcome page with the toolbar across the top. Log in with the superuser credentials you created during the install.



Your project layout

The `coffeesite/` directory now contains a working Django project:

```
coffeesite/
    LICENSE
```

(continues on next page)

(continued from previous page)

```
    README.md
    db.sqlite3
    coffeesite/
        static/
        templates/
            base.html
        __init__.py
        asgi.py
        settings.py
        urls.py
        wsgi.py
    manage.py
    requirements.in
```

You can delete or replace `LICENSE` and `README.md` for your own project. `requirements.in` is where you add new dependencies — we will add one in chapter 3.

The tutorial assumes:

- the project package is called `coffeesite`,
- a Django app called `coffeeshop` that you will create in chapter 3,
- `DEBUG = True` while you work.

Want to install by hand?

If you'd rather see every line of settings that the `djangocms` shortcut writes for you, work through *Install django CMS manually* ("Install django CMS manually") instead. You will arrive at the same place; the rest of this tutorial works identically with either install path.

In the next chapter you will create your first CMS page.

Your first page

You will create a CMS page, drop a text plugin onto it, and publish it. No code in this chapter — only the django CMS frontend editing interface.

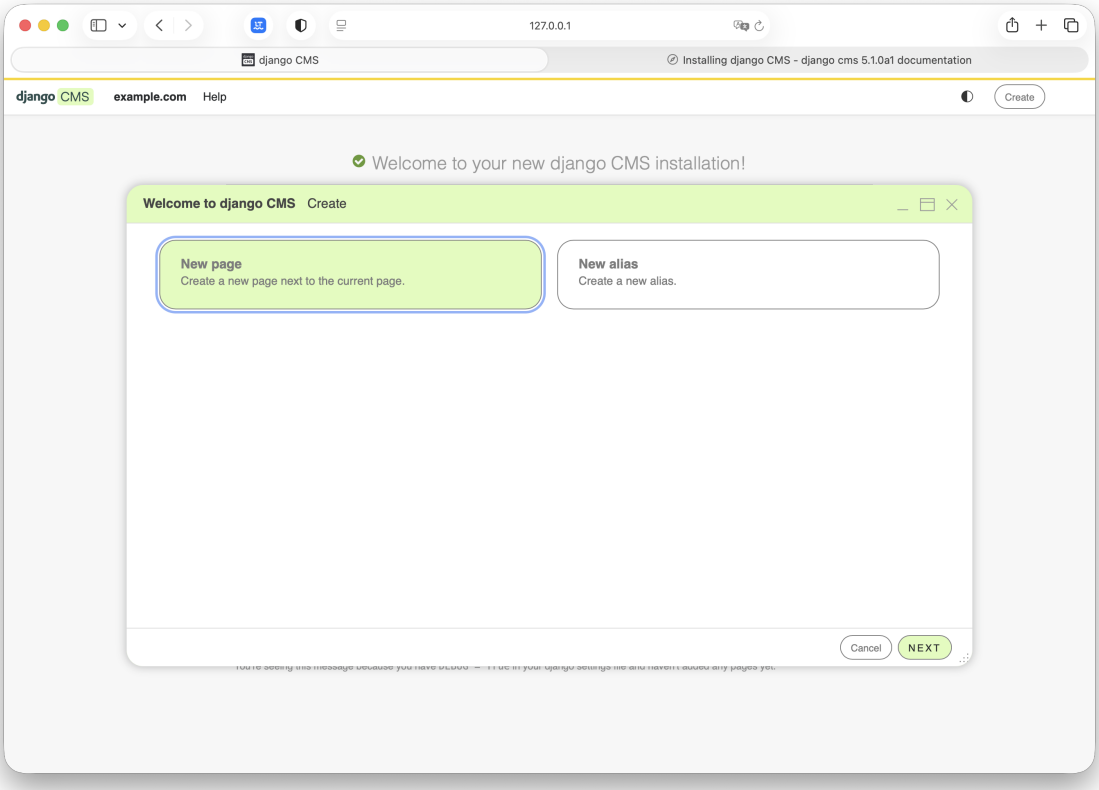
Goal

At the end of this chapter, opening `http://localhost:8000/` in a private browser window (logged out) shows a *Home* page, and `http://localhost:8000/about/` shows a second *About* page — each with a heading and a paragraph that you wrote.

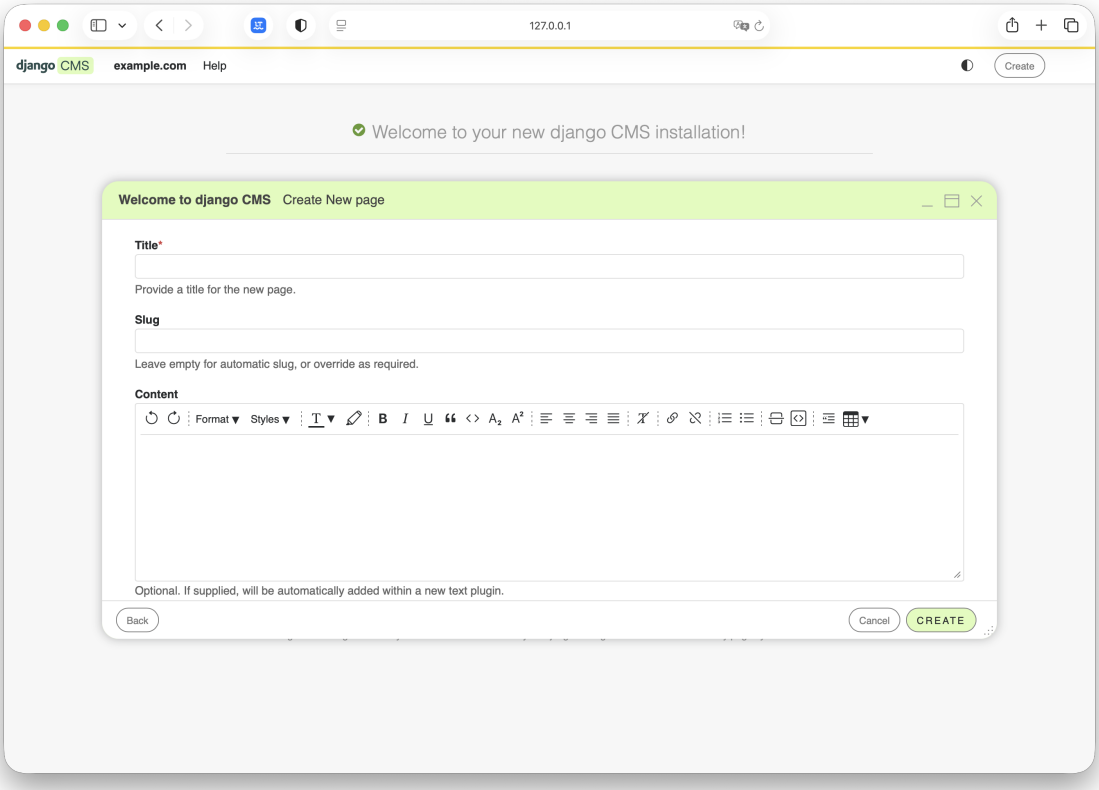
1. Create your first page

You should be logged in as the superuser created during install. The django CMS toolbar is visible at the top of the screen.

1. In the toolbar, click **Create**. The wizard opens and asks what you want to create.



2. Choose **New page** and click **Next**. Set the title to `Home`. Leave the **Content** field empty — we will add the content from the structure board in the next step. Click **Create**.



You are now on the new page in *edit mode*. The address bar shows an internal editing URL (something like `/admin/cms/placeholder/object/.../edit/...`) rather than the page's own address — that is normal while you are editing.

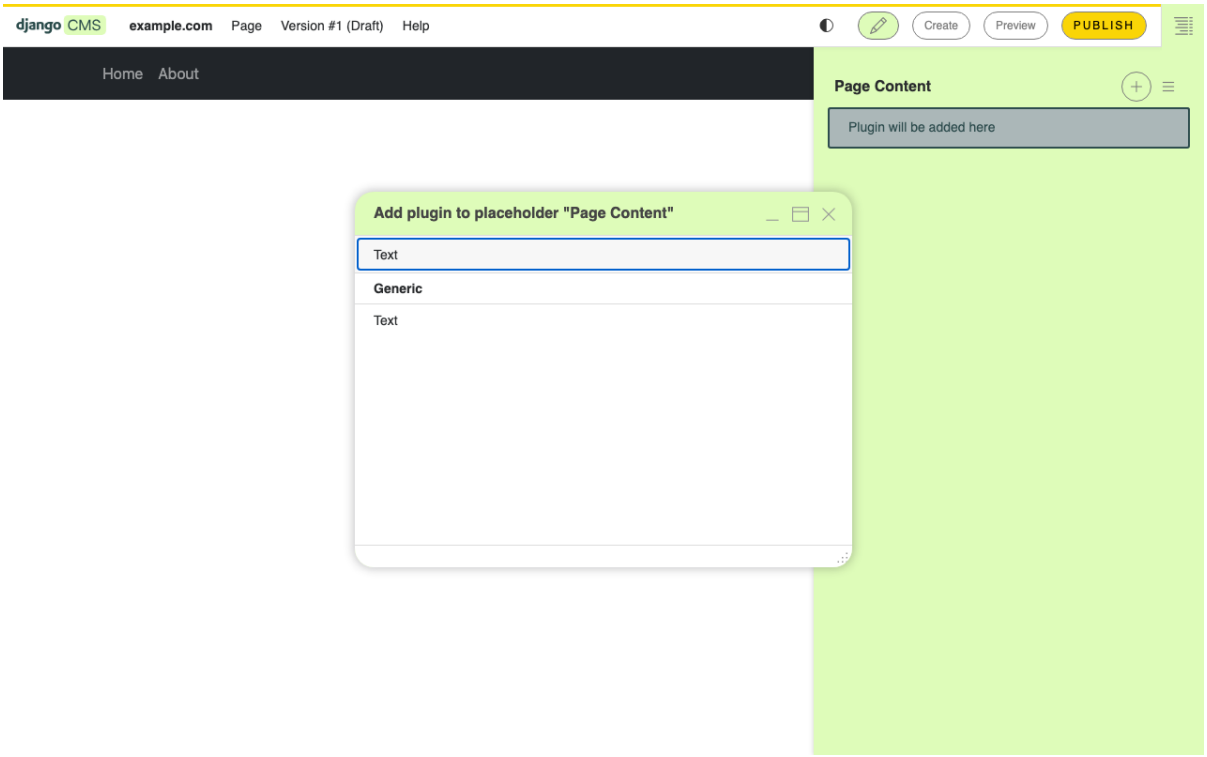
> **Note**
>
> The **first page you create automatically becomes the site's homepage**, so it is served at the site root (`http://localhost:8000/`) regardless of its slug. You can change which page is the homepage later from **Pages…** → the page's **three dots** → **Set as home**.

2. Add some content

The page is using the default template, which exposes a single *placeholder* called `Page Content` in the middle of the page.

1. Click the toggle on the top right of the page to open the structure board.
2. The structure board shows the page's single placeholder, called `Page Content`. Click the **plus button** next to it to add a plugin.
3. The plugin selector appears. Choose **Text**.
4. Type a heading and a short paragraph in the rich-text editor.
5. Click **Save** to commit changes to the page.

The page now shows your text.

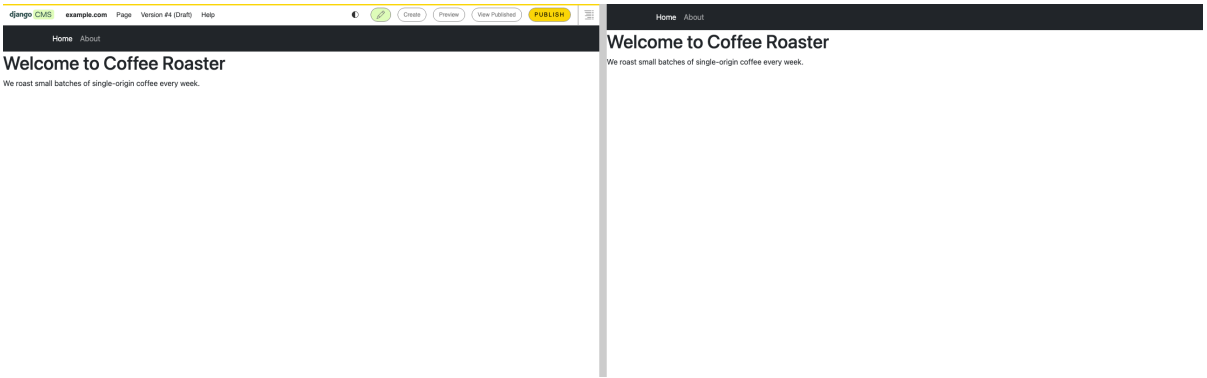


3. Publish the page

So far the page only exists in *draft*. Anonymous visitors cannot see it.

1. In the toolbar, on the right side click **Publish**.
2. Open `http://localhost:8000/` in a private browser window or a different browser where you are not logged in.

You should see your Home page. If you see a 404 instead, you forgot the publish step.



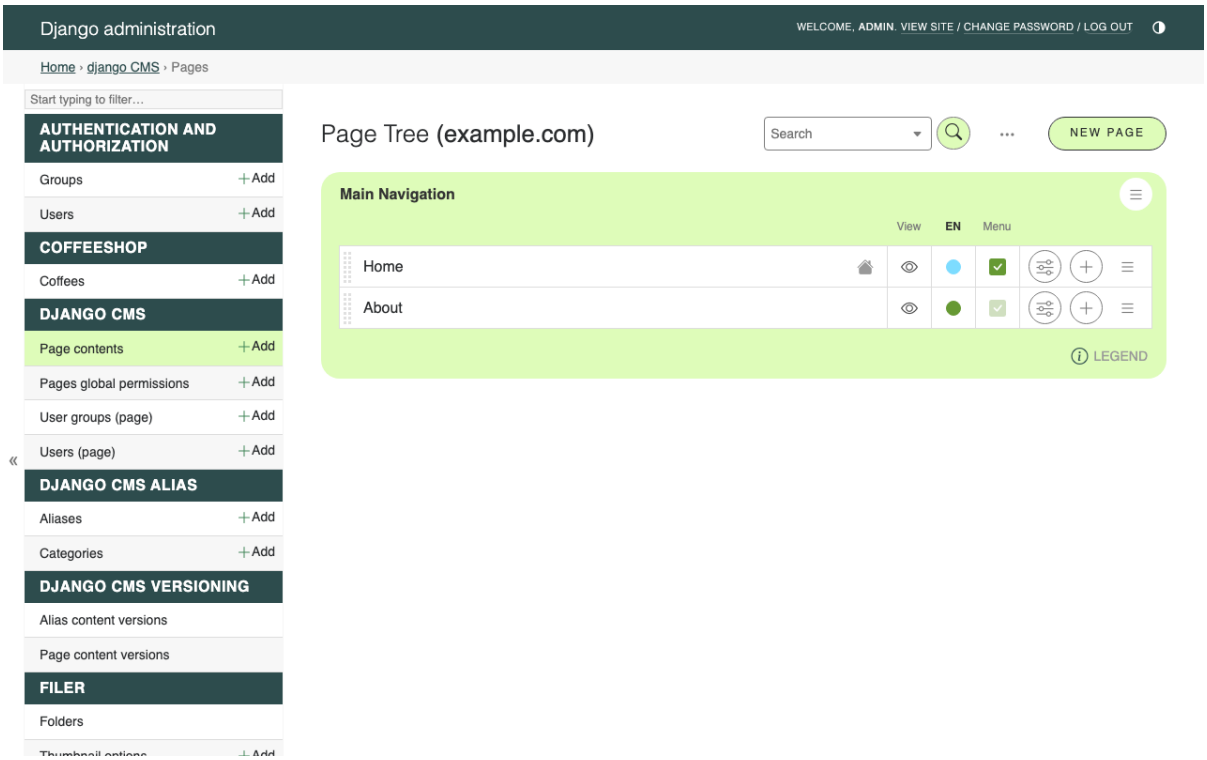
4. Create a second page

Now add a second page that lives at its own URL.

1. In the toolbar, click **Create** → **New page**.
2. Set the title to `About` and leave the **Content** field empty. Click **Create**.

3. Add a **Text** plugin to its `Page Content` placeholder, just as you did for the home page, and write a heading and a paragraph.
4. **Publish** the page.

Because the homepage already exists, this second page is served at its own slug. Open `http://localhost:8000/about/` in a private window — you should see your About page.



Now `http://localhost:8000/` serves the *Home* page, and `/about/` serves the About page.

What just happened

You created two CMS *pages*, each of which contained one *placeholder*, each of which held one *plugin*. That is the core data model of django CMS:

Page → placeholders → plugins

You will see those three words in every later chapter.

For the conceptual story behind pages and placeholders, see *Plugins* and *Publishing*. For the toolbar's full menu structure, see *Toolbar*.

Going further

- *How to serve multiple languages* — serve the same pages in multiple languages.
- *How to manage caching* — page caching for production.

In the next chapter we will swap the default template for one of our own and learn how placeholders are declared.

Templates and placeholders

So far the CMS rendered your pages using the project's default `base.html`, which simply extends a template from djangocms-frontend and exposes a single `Page Content` placeholder. In this chapter you will replace that base template with your own, defining two placeholders — `Header` and `Body` — so editors can manage a header strip *and* a body separately.

Goal

At the end of this chapter, the homepage uses a template you wrote and exposes two placeholders: `Header` and `Body`. Editors can place plugins into either one.

1. Find the project's base template

The `djangocms` command already created a base template for you and told both Django and the CMS about it:

- The template lives at `coffeesite/templates/base.html`.
- `settings.py` already points `TEMPLATES` → `DIRS` at `coffeesite/templates`, so Django finds it. You do **not** need to create a new directory or edit `DIRS`.
- `settings.py` already registers it for the CMS:

  ```
  CMS_TEMPLATES = [
      ("base.html", "Standard"),
  ]
  ```

  The second entry of each tuple is the human-readable label shown in the toolbar. Rename `"Standard"` to something like `"Coffee Roaster - base"` if you like; there is no need to add a new entry.

Open `coffeesite/templates/base.html`. Out of the box it is a two-line stub that extends a template from djangocms-frontend — that is where the single `Page Content` placeholder from the previous chapter comes from:

```
{# Replace this with your base template #}
{% extends "bootstrap5/base.html" %}
```

We will replace this stub with our own markup.

2. Write your own base template

Replace the contents of `coffeesite/templates/base.html` with:

```
{% load cms_tags sekizai_tags %}
<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>{% page_attribute "page_title" %} - Coffee Roaster</title>
    {% render_block "css" %}
</head>
<body>
    {% cms_toolbar %}

    <header>
        {% placeholder "Header" %}
    </header>
```

(continues on next page)

(continued from previous page)

```
    <main>
        {% placeholder "Body" %}
    </main>

    {% render_block "js" %}
</body>
</html>
```

A few things to notice:

- `{% placeholder "Name" %}` is what creates a placeholder. The name is what editors see in the toolbar.
- `{% cms_toolbar %}` is required for the toolbar to render — without it, you cannot edit.
- `{% render_block "css" %}` and `{% render_block "js" %}` come from django-sekizai. Plugins ship their own CSS and JavaScript; when several plugins on the same page need the same asset, sekizai collects them so each block is rendered exactly once, in the right place in the document. Every django CMS template needs the `"css"` block in `<head>` and the `"js"` block just before `</body>`.

For everything `{% placeholder %}` accepts, see *Placeholders*. For the conceptual story, see *Plugins*.

3. Reload the page

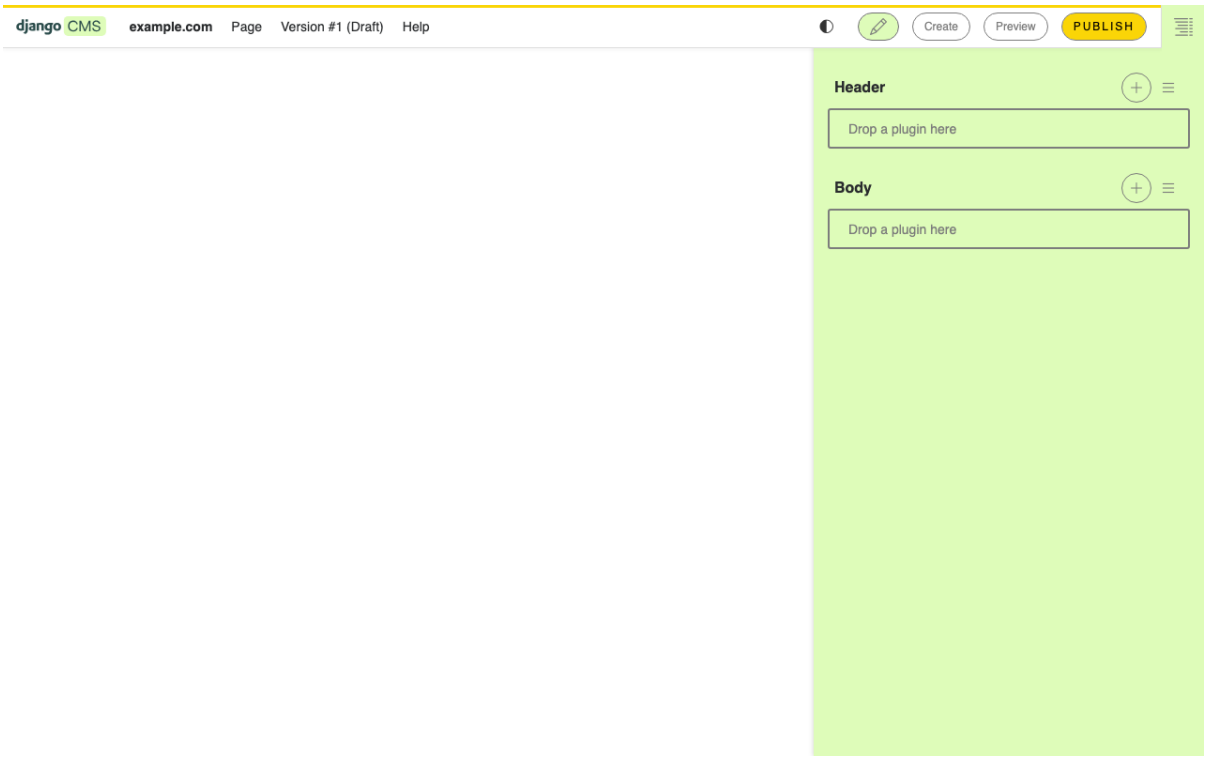
The home page already uses `base.html`, so there is no template to switch — your new markup takes effect as soon as the file is saved.

1. Reload the *Home* page in your browser in edit mode.
2. You will see two empty placeholders labelled `Header` and `Body`.

If the new placeholders do not appear, restart `runserver` and reload the page.

Note

The text you added to `Page Content` in the previous chapter is still stored, but it no longer appears: the new template does not include a `Page Content` placeholder, so its plugins are not rendered.



4. Fill the placeholders

1. Drop a **Text** plugin into `Header` and type the site name — "Coffee Roaster".
2. Drop a **Text** plugin into `Body` and write a short welcome paragraph.
3. **Publish** the page.

Visit the homepage in a private window. You should see a header strip above the body content.

What just happened

You now own the markup. Two new ideas appeared:

- A **template** declares which placeholders exist on a page. Different templates can declare different placeholders.
- **CMS_TEMPLATES** is the list of templates editors can pick from.

Anything you would do in a normal Django template — `{% block %}` inheritance, `{% include %}`, `{% url %}` — works inside a CMS template. The only CMS-specific tags you need for now are `{% cms_toolbar %}` and `{% placeholder %}`.

A reusable region with `static_alias`

If you want a region whose content is *the same on every page* (a footer, for example), use `{% static_alias %}` instead of `{% placeholder %}`. It works similarly for editors, but the content is stored once and reused everywhere:

```
<footer>
    {% static_alias "site_footer" %}
</footer>
```

Add that to `base.html` if you like. Do not forget to add `{% load djangocms_alias_tags %}` at the top of the file to let Django know about the static alias tag. The full mechanics live in *How to work with templates*.

Going further

- *How to work with templates* — multiple templates, inheritance, per-page template overrides.
- *How to use placeholders outside the CMS* — placeholders outside CMS pages (e.g. on your own Django models).

In the next chapter we leave the toolbar and write our first custom plugin in Python.

A custom plugin

django CMS does not come with built-in plugins. We have been using plugins (Text, Image, etc.) provided by packages such as djangocms-text or djangocms-frontend. Standardized plugins only get you so far. In this chapter you will create a *Coffee card* plugin: a small reusable component editors can drop into any placeholder to show one of your coffees, with a name, an origin, and a price.

Goal

At the end of this chapter, the **Add plugin** menu in the toolbar contains a *Coffee card* entry. Choosing it opens a form with three fields. Saving renders a styled card in the placeholder.

1. Create the `coffeeshop` app

In your project root:

```
python manage.py startapp coffeeshop
```

Add it to `INSTALLED_APPS` in `settings.py`:

```
INSTALLED_APPS = [
    # ... existing apps ...
    "coffeeshop",
]
```

2. Define the plugin model

Plugin data is stored in the database. Django CMS plugin models inherit from *`CMSPlugin`*, **not** from `models.Model`.

Open `coffeeshop/models.py` and replace its contents with:

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


class CoffeeCard(CMSPlugin):
    name = models.CharField(max_length=80)
    origin = models.CharField(max_length=80)
    price = models.DecimalField(max_digits=6, decimal_places=2)

    def __str__(self):
        return self.name
```

Create and run the migration:

```
python manage.py makemigrations coffeeshop
python manage.py migrate coffeeshop
```

3. Register the plugin class

The model holds the data. A separate *plugin class* tells the CMS how to render it and what label to show in the menu. Plugin classes live in `cms_plugins.py`.

Create `coffeeshop/cms_plugins.py`:

```
from cms.plugin_base import CMSPluginBase
from cms.plugin_pool import plugin_pool

from coffeeshop.models import CoffeeCard


@plugin_pool.register_plugin
class CoffeeCardPlugin(CMSPluginBase):
    model = CoffeeCard
    name = "Coffee card"
    render_template = "coffeeshop/coffee_card.html"
    cache = True

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

4. Add the render template

Create `coffeeshop/templates/coffeeshop/coffee_card.html`:

```
<article class="coffee-card">
    <h3>{{ instance.name }}</h3>
    <p class="origin">{{ instance.origin }}</p>
    <p class="price">${{ instance.price }}</p>
</article>
```

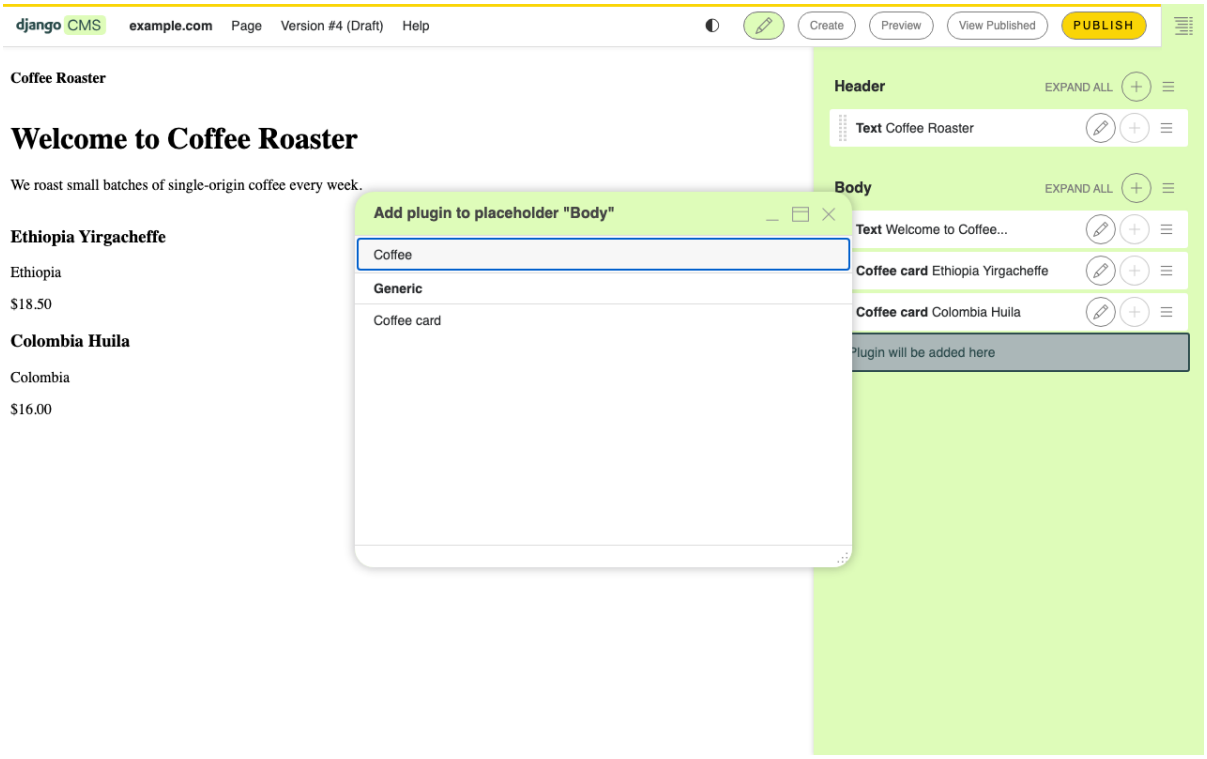
The `instance` in the template is the `CoffeeCard` row the editor created.

5. Try it

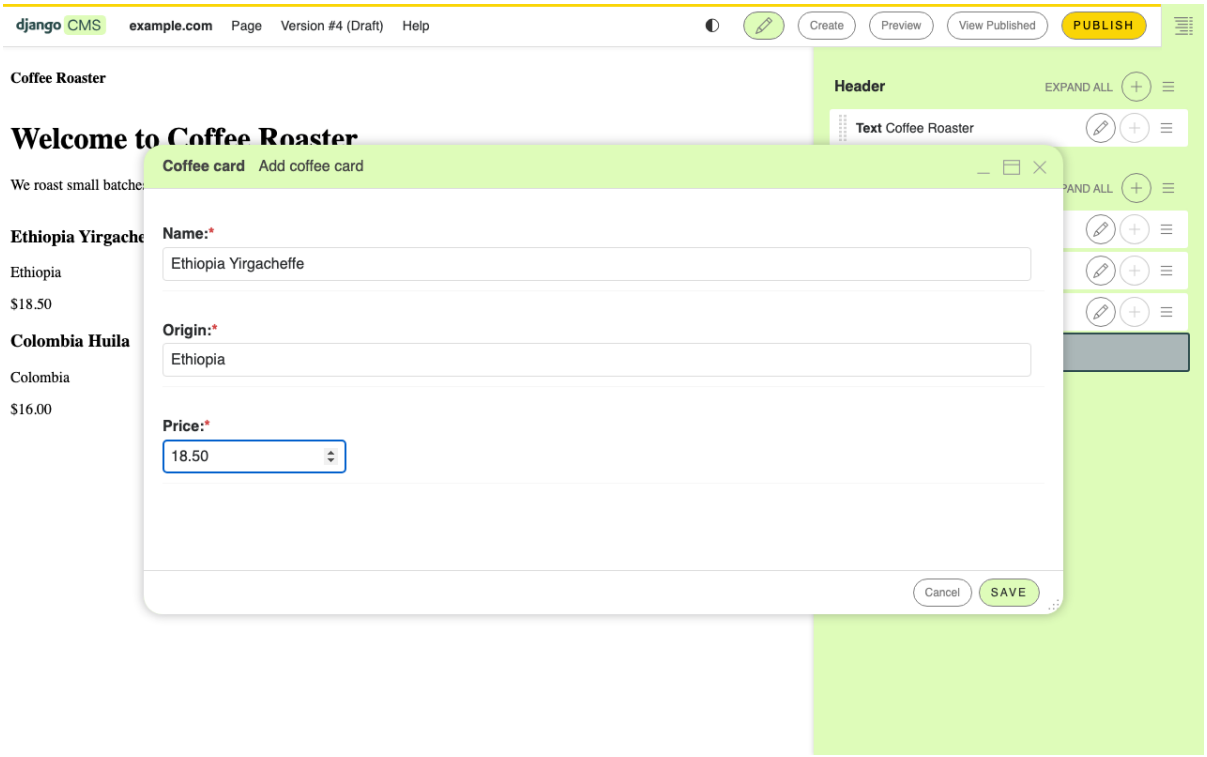
Restart `runserver` (necessary, because you added a new `cms_plugins.py`).

In your browser, open the homepage in edit mode:

1. Click into the `Body` placeholder.
2. **Add plugin** → notice the new *Coffee card* entry.
3. Fill in a name (`Ethiopia Yirgacheffe`), an origin (`Ethiopia`), and a price (`18.50`). Save.
4. The card renders in place.



Choosing *Coffee card* opens the plugin's editing form. You did not write this form — the CMS generated it from the three fields on your `CoffeeCard` model, the same way the Django admin builds forms from a `ModelAdmin`:



Drop a second card in. Drag to reorder. **Publish** the page when you are happy.

You now have a custom component that any editor can reuse across the site.

What just happened

A django CMS plugin is three small files:

1. A **model** that subclasses `CMSPlugin` — what data to store.
2. A **plugin class** registered with `plugin_pool` — what the editor sees, which template to render.
3. A **template** — what visitors see.

Every plugin in every django CMS site follows the same shape.

What we did **not** do — and where to find it when you need it:

- **Copying plugins across page versions** (`copy_relations`) — see *How to create Plugins*.
- **Plugins that contain other plugins** (allow_children) — see *How to create Plugins*.

- **Plugins with foreign keys to your own models** — same place.
- **The full plugin API surface** — see *Plugins*.

Going further

- *How to create Plugins* — every advanced plugin pattern.
- *Plugins* — why the model/view/template split, and how plugins relate to placeholders.

In the next chapter we will leave plugins behind and mount an entire Django app on a CMS page using an apphook.

Mount a Django app with an apphook

Plugins are for components dropped *into* pages. **Apphooks** are for attaching an entire Django app *to* a page, so the URL of that page becomes the entry point of the app.

In this chapter you will:

- add a `Coffee` model with list and detail views to the `coffeeshop` app,
- create an apphook that exposes those views,
- attach the apphook to a new CMS page at `/menu/`.

Goal

At the end of this chapter, `http://localhost:8000/menu/` is served by your `coffeeshop` views (not by a CMS placeholder), and each coffee links to its own detail page below `/menu/`. Editors can move the page in the page tree like any other page, and the app moves with it.

1. A catalogue model

Add this to `coffeeshop/models.py` (keep the `CoffeeCard` plugin model from the previous chapter):

```
class Coffee(models.Model):
    name = models.CharField(max_length=80)
    origin = models.CharField(max_length=80)
    price = models.DecimalField(max_digits=6, decimal_places=2)
    description = models.TextField(blank=True)

    class Meta:
        ordering = ("name",)

    def __str__(self):
        return self.name
```

Run the migration:

```
python manage.py makemigrations coffeeshop
python manage.py migrate coffeeshop
```

Register the model in `coffeeshop/admin.py` so you can add a few rows:

```
from django.contrib import admin
from coffeeshop.models import Coffee

admin.site.register(Coffee)
```

Open `/admin/` and add three or four coffees.

2. A list view

Create `coffeeshop/views.py`:

```
from django.views.generic import ListView
from coffeeshop.models import Coffee


class CoffeeListView(ListView):
    model = Coffee
    template_name = "coffeeshop/coffee_list.html"
    context_object_name = "coffees"
```

And its template at `coffeeshop/templates/coffeeshop/coffee_list.html`:

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

{% block content %}
    <h1>Our coffees</h1>
    <ul class="coffee-list">
        {% for coffee in coffees %}
            <li>
                <h2>{{ coffee.name }}</h2>
                <p>{{ coffee.origin }} · ${{ coffee.price }}</p>
                <p>{{ coffee.description }}</p>
            </li>
        {% endfor %}
    </ul>
{% endblock %}
```

(If your `base.html` from chapter 2 has no `{% block content %}`, add one wrapping the `{% placeholder "Body" %}` line.)

A URL conf for the app at `coffeeshop/urls.py`:

```
from django.urls import path
from coffeeshop.views import CoffeeListView

app_name = "coffeeshop"

urlpatterns = [
    path("", CoffeeListView.as_view(), name="list"),
]
```

Do **not** add this to your project's main `urls.py`. The apphook will do that for us.

3. A detail view

Each coffee should get a page of its own. Add a `DetailView` next to the list view in `coffeeshop/views.py`:

```
from django.views.generic import DetailView, ListView
from coffeeshop.models import Coffee
```

(continues on next page)

(continued from previous page)

```
class CoffeeListView(ListView):
    model = Coffee
    template_name = "coffeeshop/coffee_list.html"
    context_object_name = "coffees"


class CoffeeDetailView(DetailView):
    model = Coffee
    template_name = "coffeeshop/coffee_detail.html"
    context_object_name = "coffee"
```

Create its template at `coffeeshop/templates/coffeeshop/coffee_detail.html`:

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

{% block content %}
    <h1>{{ coffee.name }}</h1>
    <p>{{ coffee.origin }} · ${{ coffee.price }}</p>
    <p>{{ coffee.description }}</p>
    <p><a href="{% url 'coffeeshop:list' %}">Back to all coffees</a></p>
{% endblock %}
```

Route it in `coffeeshop/urls.py`:

```
from coffeeshop.views import CoffeeDetailView, CoffeeListView

urlpatterns = [
    path("", CoffeeListView.as_view(), name="list"),
    path("<int:pk>/", CoffeeDetailView.as_view(), name="detail"),
]
```

Finally, make each coffee in the list link to its detail page. In `coffeeshop/templates/coffeeshop/coffee_list.html`, change the name line to:

```
<h2><a href="{% url 'coffeeshop:detail' coffee.pk %}">{{ coffee.name }}</a></h2>
```

4. The apphook

Create `coffeeshop/cms_apps.py`:

```
from cms.app_base import CMSApp
from cms.apphook_pool import apphook_pool


@apphook_pool.register
class CoffeeshopApphook(CMSApp):
    app_name = "coffeeshop"
    name = "Coffee shop catalogue"

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

`app_name` is the URL namespace the apphook installs into. Because `coffeeshop/urls.py` also declares `app_name = "coffeeshop"` and gives the list view `name="list"`, you can reverse it from anywhere in the project as `{% url "coffeeshop:list" %}` — even though the URL itself is decided by whichever CMS page you attach the apphook to.

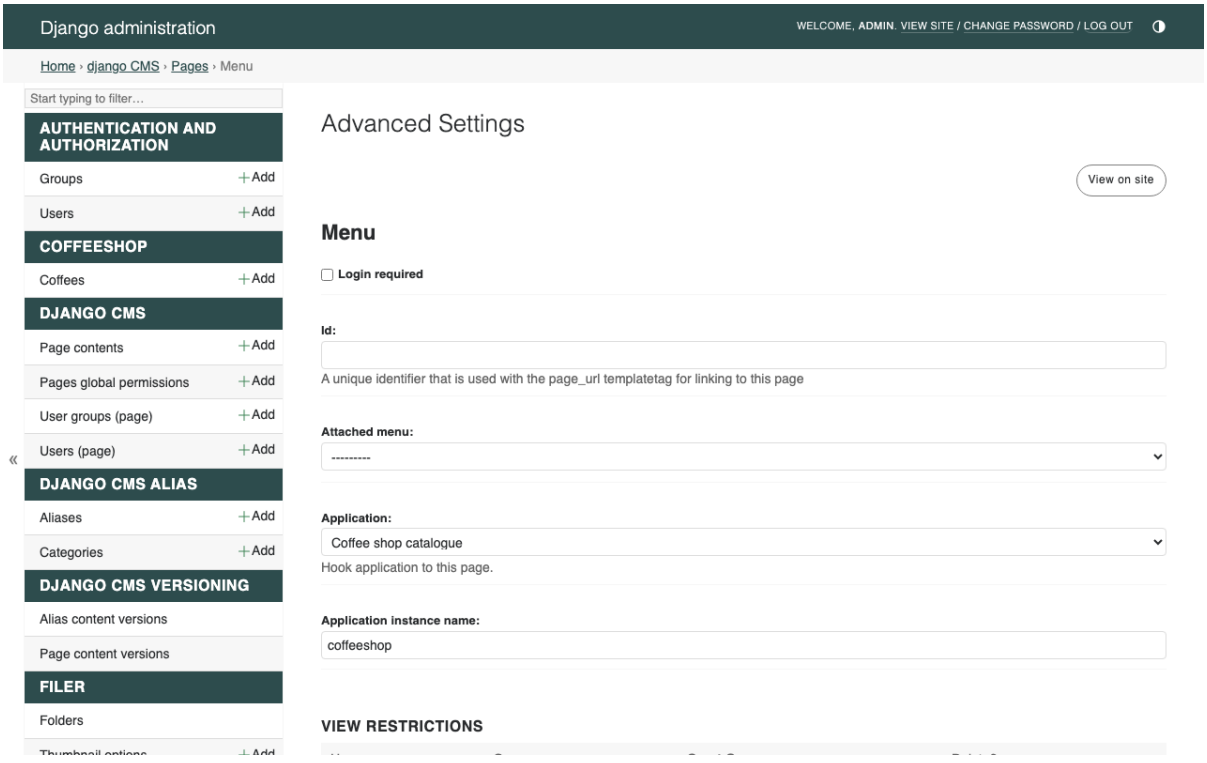
Restart `runserver` (apphooks are loaded at startup).

> **Tip**
>
> If you find yourself attaching, moving, or removing apphooks while you work, add `cms.middleware.utils.ApphookReloadMiddleware` at the **top** of `MIDDLEWARE` in `settings.py`. The CMS then reloads the URL conf in-process when an apphook changes, and you no longer need to restart `runserver` for each edit. (The `djangocms` quickstart already includes it.)

5. Attach the apphook to a page

In the toolbar:

1. **Create** → **New page**. Title: `Menu`. Click **Create**.
2. **Page** → **Advanced settings…**
3. In the **Application** dropdown, choose *Coffee shop catalogue*.
4. Save.
5. **Publish** the page.



Visit `http://localhost:8000/menu/` — you should see your coffee list, rendered by `CoffeeListView` extending `base.html`. Click a coffee's name: its detail page is served at a URL like `/menu/2/`, below the page's URL.

You can rename the page slug, move the page in the tree, or translate it; the URL of the apphook follows the page.

What just happened

You attached a Django app to a CMS page. The CMS:

- finds the page that has the apphook,
- prepends that page's URL to your app's `urlpatterns`,
- hands every request below that page to your views.

That is why URL changes in the toolbar move the app with them — and why `get_urls` returns a whole URL conf rather than a single view: the list *and* every detail page below it follow the page together.

Important

Do not create CMS child pages *underneath* a page that has an apphook. The apphook owns every URL below it; child pages would never be reachable. The conceptual story is in *Application hooks ("apphooks")*.

Going further

- *How to create apphooks* — apphooks with custom URL patterns, `permissions`, and reloading without restarting.
- *How to manage complex apphook configurations* — multiple instances of the same apphook on different pages.
- *Configuring apps to work with django CMS* — the full `CMSApp` API.

One chapter left. We will give the site some style and a real navigation.

Make it look like a site

So far the site works but reads as an admin demo. In this short chapter you will add a small stylesheet, render the CMS menu as a real top navigation, and replace the unstyled body in `base.html` with a proper layout.

Goal

At the end of this chapter, an anonymous visitor lands on a homepage with a navigation bar containing **Home**, **About**, and **Menu**. Pages share a visual style. The coffee cards look like cards.

1. A minimal stylesheet

Create `coffeeshop/static/coffeeshop/site.css`:

```
:root {
    --ink: #2b1d12;
    --paper: #faf6f1;
    --accent: #a4673e;
}

body {
    font-family: system-ui, sans-serif;
    margin: 0;
    background: var(--paper);
    color: var(--ink);
    line-height: 1.5;
}

header.site-header {
    background: var(--ink);
    color: var(--paper);
    padding: 1rem 2rem;
    display: flex;
    align-items: center;
    gap: 2rem;
}

header.site-header a {
    color: inherit;
    text-decoration: none;
}

nav.site-nav ul {
    list-style: none;
    display: flex;
    gap: 1.5rem;
    margin: 0;
    padding: 0;
}

main {
    max-width: 56rem;
    margin: 2rem auto;
    padding: 0 1rem;
}

.coffee-card {
    border: 1px solid var(--accent);
    border-radius: 6px;
    padding: 1rem 1.25rem;
    margin: 1rem 0;
    background: white;
}
```

(continues on next page)

(continued from previous page)

```
.coffee-card h3 {
    margin: 0 0 .25rem;
}

.coffee-card .price {
    font-weight: bold;
    color: var(--accent);
}
```

You do not have to type this — copy any starter stylesheet you like. The point is that styling is ordinary CSS.

Run `python manage.py collectstatic --noinput` if your project serves static files via Django's staticfiles app. During development with `DEBUG = True` you usually do not need to.

2. Render the navigation from CMS pages

The CMS menu is a *template tag*. Edit `coffeesite/templates/base.html` to import it and use it inside the header.

```
{% load cms_tags sekizai_tags menu_tags static %}
<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>{% page_attribute "page_title" %} - Coffee Roaster</title>
    <link rel="stylesheet" href="{% static 'coffeeshop/site.css' %}">
    {% render_block "css" %}
</head>
<body>
    {% cms_toolbar %}

    <header class="site-header">
        <a href="/"><strong>Coffee Roaster</strong></a>
        <nav class="site-nav">
            <ul>{% show_menu 0 1 100 100 %}</ul>
        </nav>
    </header>

    <main>
        {% block content %}
            {% placeholder "Body" %}
        {% endblock %}
    </main>

    {% render_block "js" %}
</body>
</html>
```

The four numbers on `{% show_menu %}` control how deep and how many levels the menu shows. Defaults are sane. The full list of menu tags is in *User site navigation*.

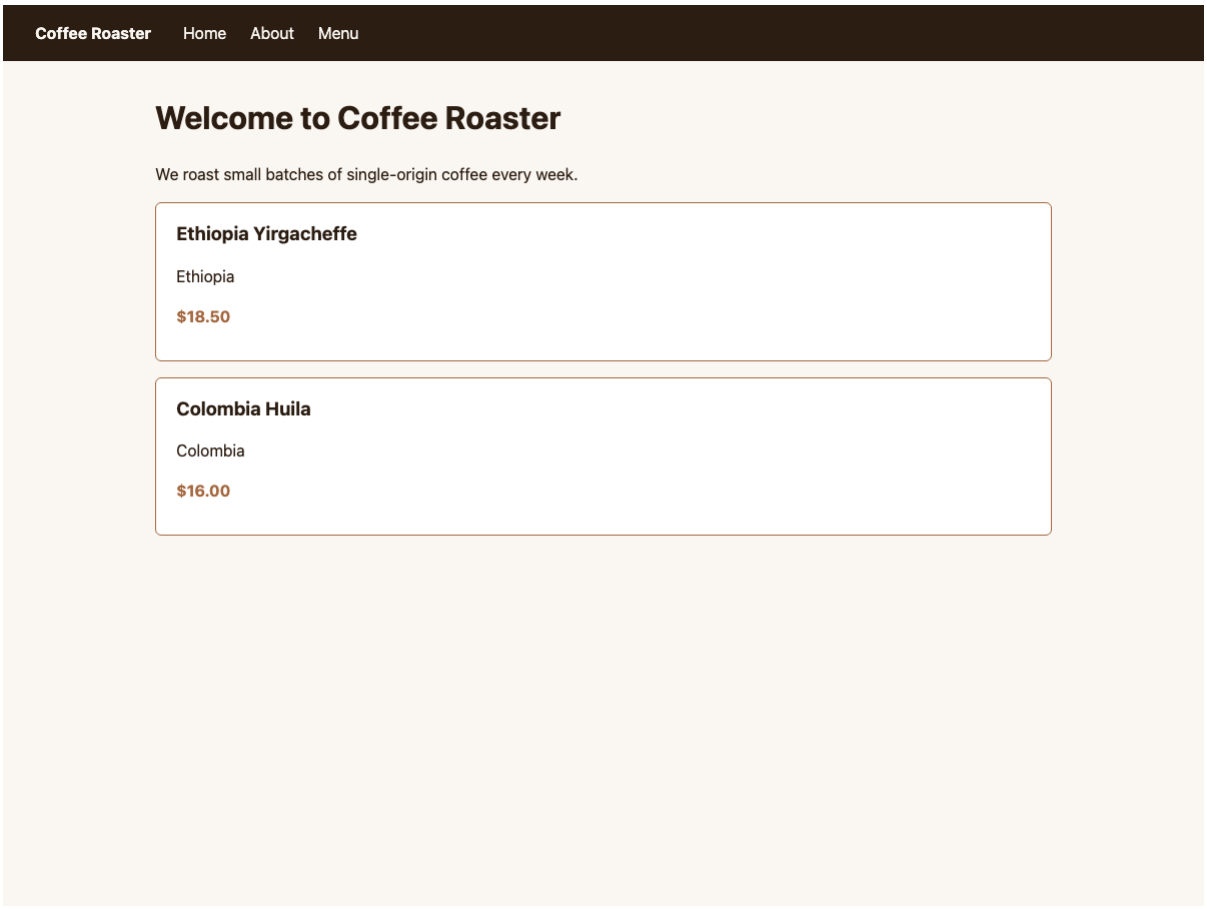
The default `menu/menu.html` template renders `<li>` items — only nested levels bring their own `<ul>` — so you provide the outer `<ul>` yourself, as in the snippet above. The CSS above styles the top level into a horizontal bar.

3. Drop the placeholder `Header`

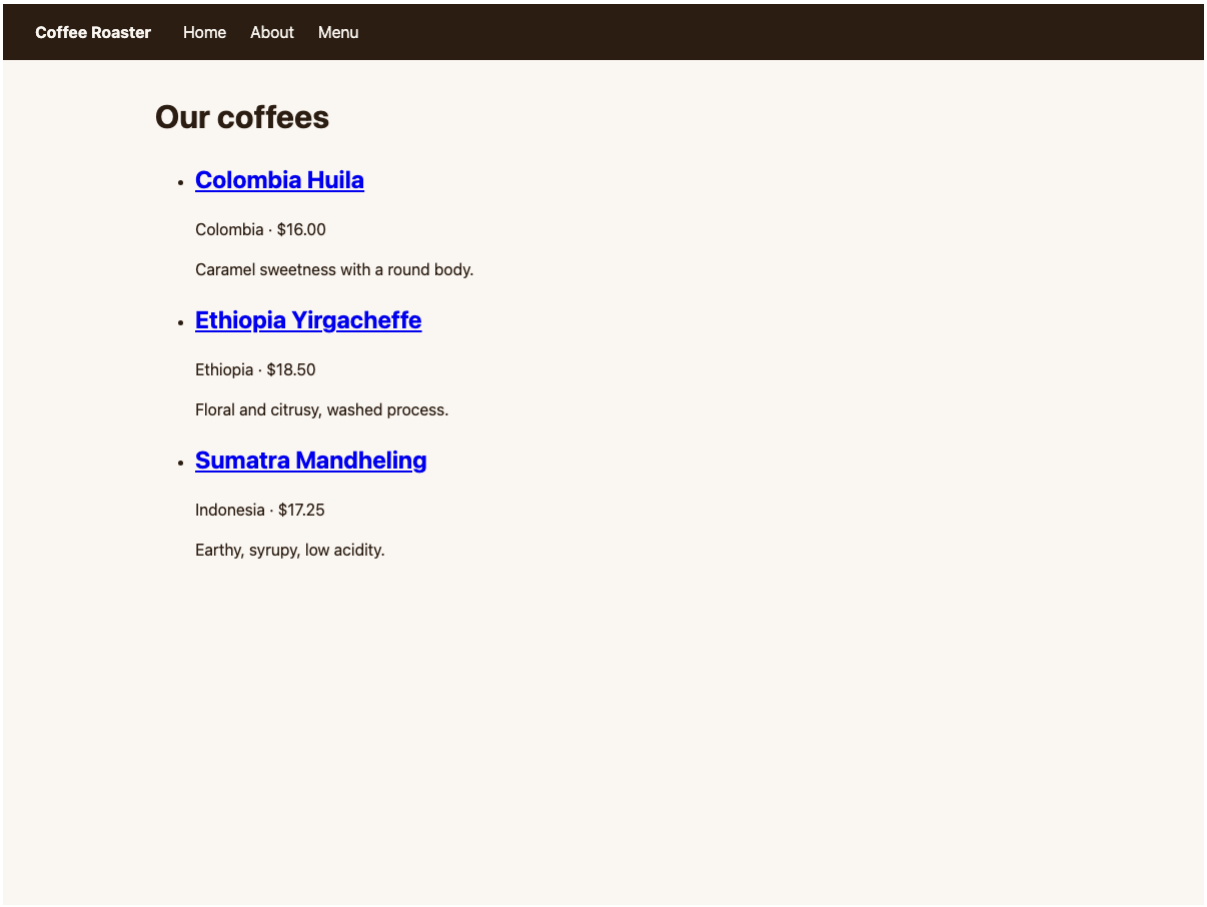
The navigation is now part of `base.html` itself, not a placeholder. You can either delete the `{% placeholder "Header" %}` line from chapter 2 or leave it for future per-page banner content. For this tutorial, delete it.

Restart `runserver`. Refresh `/` in a private window. You should see:

- a dark header with the site name and three menu items,
- styled coffee cards on the homepage,
- the catalogue at `/menu/` inheriting the same chrome.



The catalogue at `/menu/` gets the same treatment without any extra work: it is served by your apphook view, but its template extends `base.html`, so the new header, navigation and stylesheet apply there too:



If the menu shows pages you wanted hidden, mark them as **not in navigation** in **Page** → **Advanced settings** → *Menu*.

What just happened

Two CMS-specific pieces appeared:

- `{% show_menu %}` reads the page tree and renders a navigation rooted at the start of it. Pages are added, removed, and reordered in the toolbar; the menu reflects it without any code change.
- The CMS menu is **also** extendable from Python — you can inject nodes that do not correspond to CMS pages (e.g. blog post entries). See *How to customise navigation menus*.

Going further

- *User site navigation* — every menu template tag and option.
- *How the menu system works* — soft roots, modifiers, and the full mental model.
- *How to customise navigation menus* — adding non-CMS pages to the navigation.

You have built a working django CMS site. The last short chapter points you at the rest of the documentation.

Where to go next

The tutorial is over. You can publish CMS pages, write your own plugins, and mount Django apps via apphooks. This page is a map of where to look when you need more.

If you have a *task* in mind

Reach for *How-to guides*. The how-to guides are short, focused recipes:

- *How to manage caching* — page and plugin caching.
- *How to serve multiple languages* — serve content in multiple languages.
- *Multi-Site Installation* — run multiple sites from one project.
- *How to extend the Toolbar* — add custom buttons and menus to the toolbar.
- *How to implement content creation wizards* — create content-creation wizards for the toolbar's *Create* button.
- *How to extend Page & PageContent models* — add custom fields to pages.
- *How to run django CMS in headless mode* — use django CMS as a headless backend.
- *Install django CMS with Docker* and *Install django CMS manually* — deployment-shaped setups.

If you want to understand *why*

Reach for *Explanation*:

- *Philosophy* — design principles behind the CMS.
- *Plugins* — the model/view/template split for plugins, and how plugins compose with placeholders.
- *Application hooks ("apphooks")* — what an apphook really is.
- *Publishing* — drafts, versions, and the publishing model.
- *Permissions* — the permission system.

If you need the authoritative *API*

Reach for *Reference*:

- *Configuring django CMS* — every `CMS_*` setting.
- *Plugins* — `CMSPluginBase` and the plugin pool.
- *Configuring apps to work with django CMS* — `CMSApp`.
- *Template Tags* — every template tag the CMS exposes.
- *Command Line Interface* — `manage.py cms` subcommands.

Three topics worth exploring next

Most production sites need at least these three things. None of them are part of django CMS core, but each has an official package and a how-to.

Versioning. The core publishing flow has draft/published states. For richer editorial workflows (multiple draft versions, scheduled publishing) add djangocms-versioning.

A rich-text editor that suits your team. The CMS works with several editors; pick one and install it as a frontend integration. See *Some commonly-used plugins*.

A frontend admin you're happy with. `djangocms-simple-admin-style` provides a polished admin skin that pairs with the toolbar. The quickstart project includes it; you can adopt it in a manual install too.

Community

The friendliest place to ask questions is the django CMS Discord server.

If you find a bug, file it on GitHub. If you want to help, see *Contribute* — documentation fixes are some of the most valuable contributions and one of the easiest places to start.

What you will build

By the end of chapter 5 your project will contain:

- an `About` page and a `Home` page authored in django CMS' frontend editing interface,
- a custom plugin called *Coffee card* that editors can drop into any placeholder,
- a `coffeeshop` Django app exposing a catalogue at `/menu/`, mounted on a CMS page through an apphook,
- a navigation menu and minimal styling so the site reads as a site rather than an admin demo.

If you get stuck, the django CMS community is on Discord.

5.3.2 Explanation

This section explains the **design principles, concepts, and architectural decisions** behind django CMS.

Rather than showing *how to perform specific tasks*, these pages focus on *why django CMS works the way it does* and *how its core ideas shape the system as a whole*. They are intended to provide context and understanding that will help you make better design and implementation decisions in your own projects.

At the centre of django CMS is a deliberate philosophy: complex publishing requirements are best addressed through **small, composable building blocks** rather than a single monolithic system. The pages in this section explore how that philosophy is reflected throughout the CMS.

Philosophy

django CMS gives **editors** a toolbar on the live site where they compose pages from reusable components — drag a hero image into place, drop in a card grid, reorder the sections. It gives **developers** standard Django — models, views, templates, URL patterns — and plugs their work into that editing surface. And it gives **designers** full control over templates and CSS, unconstrained by an admin theme or a fixed set of layout options. Each role owns its surface; the CMS connects them without any one needing to understand the internals of the others.

Think of it as a box of LEGO bricks rather than a model-car kit: the kit gives you one car, while the bricks give you the parts and the rules for combining them. A few of those parts ship in the box; the rest you bring yourself or pick from a wide ecosystem of add-ons.

Everything else on this page follows from that single idea.

Three disciplines, three surfaces

django CMS treats **design, code, and content** as three distinct disciplines, owned by three different roles. The CMS is shaped around keeping those concerns separate:

- **Designers** work in templates and CSS. They define which drag-and-drop layout tools are available in the front end; the layout vocabulary is whatever the templates and CSS provide.
- **Developers** work in Django: in apps, models, plugins, apphooks, and configuration. The CMS's extensibility is exposed as Python APIs, not as a no-code builder.
- **Editors** work in the toolbar and the structure board. They compose pages by adding plugins to placeholders, never by editing templates or writing code.

Tools that blur the three roles trade flexibility and maintainability for short-term convenience; django CMS deliberately sides with the former.

A small, stable core

The core of django CMS does a deliberately limited set of things:

- manage **pages** and their hierarchical structure,
- expose **placeholders** that content objects can publish into,
- coordinate **plugins** and **apphooks** so that other Django apps can plug into the CMS,
- integrate editing into rendered pages.

Almost everything else — versioning, alias content, headless rendering, the rich-text editor, the frontend admin theme — lives in separate packages. The core stays small and stable; the surface area you adopt is the surface area you need.

For *how* pages, plugins, apphooks, and other content objects fit together, see *How django CMS is composed*.

Content-agnostic

django CMS does not assume what kind of site you are building. It does not ship a Blog model, a Product model, a News model, or a Team model. It provides the **framework** — pages, content objects, placeholders, plugins, the apphook mechanism — and leaves the domain modelling to your apps.

This is a common source of "why doesn't it just…" questions, and the answer is always the same: because every project's notion of a blog, a product, or a team is slightly different, and the CMS would rather host *your* model cleanly than impose *its* model in a way not optimal for you.

Having said that, the django CMS ecosystem includes many reusable add-ons such as djangocms-stories for blogs and news.

Alignment with Django

django CMS is designed to integrate naturally into a Django project, not to replace or wrap Django. It builds on:

- apps and URL routing,
- authentication and permissions,
- models, migrations, and views,
- the Django admin.

The result is that custom application logic and CMS-managed content coexist cleanly, and familiar Django patterns and tooling apply throughout the project. A django CMS project is a Django project first.

When django CMS may not be the right fit

A philosophy page should be honest about the shape of its own tradeoffs. django CMS is not the right tool for every site:

- **It is a developer's CMS.** Reaching a working site requires some Django code (settings, templates, sometimes a small app). If nobody on the project is comfortable doing that, a no-code site builder will be a faster path.
- **It is overkill for very small or static sites.** A two-page brochure does not benefit from the publishing model, the page tree, or the plugin system. A static site generator is lighter.
- **It does not ship ready-made themes.** Where WordPress and similar platforms offer thousands of drop-in themes, django CMS expects you to bring your own frontend. Add-ons such as `djangocms-frontend` provide building blocks for established CSS frameworks (especially Bootstrap), and you are free to integrate any

third-party theme or design system you like, but the templating and styling work is yours to do. On the other hand: there are no limits to design creativity or frontend frameworks.

- **Major upgrades take work.** django CMS follows Django's LTS cadence, which limits surprises, but moving a project across a major CMS version still requires planning — especially if you rely on add-ons that lag behind.
- **The surface area is large.** Versioning, multilingual content, multi-site, headless mode, the permission model — each is powerful and each takes time to learn. Small teams should adopt them incrementally rather than all at once. We believe it to be an exciting learning curve, but it is a curve nonetheless.

If your project's centre of gravity is a Django application that happens to need a few editable pages, or if you want editors to compose pages from reusable components without being exposed to your application's internals, django CMS is squarely in its element.

Implications for projects

The philosophy above leads to several practical outcomes:

- django CMS can be introduced **incrementally** into an existing Django project — you do not have to rebuild around the CMS.
- Apps and CMS content can **evolve independently**, because the integration points (plugins, apphooks, content objects) are narrow and explicit.
- Long-term maintenance and upgrades are easier to manage, because the small core changes slowly and add-ons can usually be swapped.
- Editors gain flexible content tools **without being exposed to application complexity** they do not need.

The rest of the Explanation section explores how these principles play out in practice — see *How django CMS is composed* next, then *Publishing*, *Permissions*, and *Serving content in multiple languages* for the cross-cutting concerns.

How django CMS is composed

A django CMS site is assembled from three building blocks: **content objects** (of which the **page** is the most prominent), **plugins**, and **apphooks**. Each has a single, well-defined job. Most decisions you make while building a site are decisions about *which of the three to reach for*.

This page describes the three building blocks side by side, the rules that govern how they fit together, and the most common decision — plugin vs apphook — that beginners ask about.

For the internals of each piece, follow the cross-links:

- *Content objects: the grouper / content pattern* — the grouper / content pattern every content object follows
- *Plugins*
- *Application hooks ("apphooks")*
- *Publishing*
- *How the menu system works*

The three building blocks

Block	What it is	What it owns
Content object	A thing that holds editable content and exposes placeholders for editors to fill. **Pages** are the most common (and most powerful) kind — they live in the page tree, carry URLs, drive menus, and carry permissions. Other kinds include **aliases** (reusable content embedded via the Alias plugin) and content objects defined by add-on apps (e.g. blog posts, catalogue items).	Its placeholders, its translations, and — for pages — its slug, its template, and its position in the page tree.
Plugin	A reusable content component an editor can drop into a placeholder.	The data the editor fills in (via its model) plus the template that renders it.
Apphook	The standard way to mount a Django application onto the page tree. The application's URLs (and any content objects it defines) take their prefix from the CMS page that anchors them.	All URLs at and below the page it is attached to.

The content object is the **unit of editable content**. The plugin is the **unit of composition inside it**. The apphook is **how other content joins the page tree**.

How they fit together

The relationships are simple, but easy to confuse if you only read about each in isolation:

```
Page tree
 │
 └── Page  (a content object: owns placeholders, lives in the tree)
      │
      ├── Placeholder  (declared by the page's template)
      │    ├── Plugin
      │    │    └── Plugin (nested, if the parent allows it)
      │    └── Plugin
      │
      └── Apphook  (optional - attaches a Django app at this URL)
           │
           └── App-defined content objects
                │    (e.g. blog posts, products, polls)
                └── Placeholders
                     └── Plugins

Outside the page tree:

Alias  (a reusable content object embedded via the Alias plugin)
 └── Placeholders
      └── Plugins
```

A few rules govern the picture:

- **Content objects own placeholders. Placeholders contain plugins. Plugins compose.** This is true for every kind of content object — pages, aliases, and app-defined objects alike.
- **The page tree is what gives pages URLs**, what makes them appear in menus, and what carries the per-page permission model. Content objects that are *not* pages get their URLs by being mounted on the tree through an apphook, or by providing their URLs through their own `get_absolute_url()` method.
- **Plugins live in placeholders, never standalone.** Even when the same plugin appears on many content objects, each appearance is a separate plugin *instance* with its own configuration.
- **Plugins can contain other plugins** when the parent plugin declares `allow_children = True`. This is how composite layouts (rows, columns, accordions) are built.
- **An apphook is meaningless without a page to attach it to.** It is not a setting; it is a binding made on a CMS page in *Advanced settings*. The page provides the URL prefix; the app provides everything below.

Plugin or apphook? A decision aid

Reframed: the question is really *where does this content live?* If it fits inside an existing placeholder on someone else's page, it is a plugin. If it is its own kind of thing — with its own list view, detail view, and URL — it is an app, mounted via an apphook.

You want…	Reach for…	Why
A reusable content component the editor drops into any placeholder.	**Plugin**	Plugins are the unit of composition inside a placeholder.
A whole sub-application (blog, catalogue, polls, search) with list and detail views.	**Apphook**	Apphooks own a URL prefix and bring their own content objects.
A page where everything is editor-composed.	Page + plugins	The default flow for a page content object.
A page whose body is driven by Django views (list, detail, search results).	Page + apphook	The page provides the URL, the app provides the views and (if it has them) its own content objects.
Editors to choose *which records* show up in a list embedded on a page.	**Plugin** (with a model that references your records)	The component lives on a page; only its data lives in your app's models.
Editors to *move* a sub-site to a different URL by moving a page.	**Apphook**	The views and content objects are yours; the URL belongs to the page.
A teaser of upcoming events on the homepage *and* the full events sub-site at `/events/`.	**Both** — plugin for the teaser, apphook for the sub-site	Common combination. The plugin renders a small summary; the apphook owns `/events/` and below.

When the line blurs

A few real cases trip people up. None of them break the model; they just look like edge cases.

A plugin that needs its own detail URL. A "product card" plugin that links to `/products/42/` is still a plugin (it lives in a placeholder), but the detail URL must come from somewhere. The idiomatic answer is to **put the detail view in an apphook** mounted on a CMS page, so the URL is editor-controllable. If you wire the view into the project's root `urls.py` instead, the URL works but is fixed in code and not controlled by the editor.

Many copies of "the same" plugin on one page. Each is an independent plugin *instance* with its own configuration. They share the same model and template; they do not share data.

A page that has both placeholders and an apphook. The page's own URL (`/events/`) renders its placeholders normally; URLs *below* (`/events/2026-summit/`) are handed to the apphook. Child CMS pages under an apphooked page are *not* reliably reachable, because the apphook owns that URL space. The URLs might resolve to the app hook instead of a child page, and the child page might not be reached at all.

The same Django app mounted on several pages. Each mount point is independent. If editors should configure each mount differently (different category, different feed), the app needs an *apphook configuration*.

Reusing the same content across pages without copying it. That's what an **alias** is for: a content object that lives outside the page tree and is embedded in placeholders via the Alias plugin. A footer block, a promotional banner, or a "current opening hours" strip are typical examples.

Where to go next

- *Plugins* — what a plugin really is, what files it consists of, and how to think about plugin models.
- *Application hooks ("apphooks")* — what an apphook is, what it changes about URL handling, and the apphook configuration story.
- *Publishing* — how the publishing model interacts with every content object on the page (an unpublished page hides its plugins *and* takes its apphook offline).
- *How the menu system works* — how pages, apphooks, and custom menu code combine to produce the navigation editors see.

Content objects: the grouper / content pattern

Suppose you are building a site that needs pages in three languages, with draft/publish workflow, and a blog that inherits the same behaviour. In a typical Django project you would model each concern separately — translation tables, version fields, publish flags — and wire them together by hand.

django CMS solves this once, at the architecture level. **Every editable thing is a content object** — a page, an alias, a blog post, a product listing. Each is stored as two cooperating objects: a *grouper* that holds the long-lived identity (what it is, where it lives in the tree), and one or more *content* rows that hold the editable state (the title, the body, the template — per language, per version). Versioning, translations, and permissions plug into this split through contracts. Your model gets them without the CMS needing to know what your model is.

If you have only ever used Django models that put everything in one table, the pattern looks like one extra hop. It is. The hop is what makes the CMS treat your blog post and a CMS page as the same kind of thing — both get the draft/publish toolbar, both get translated through the same mechanism, both participate in the same permission model.

The two parts

Part	What it holds	Example fields (for a Page)
Grouper	The long-lived **identity** of the content object. One row per "thing". Survives translations, versions, and edits.	Site, tree position, `is_home`, `application_urls` (the apphook binding), `login_required`.
Content	The **editable state** of the grouper, for one combination of *language* × *version* × *any other axis* a versioning package cares about. Many rows per grouper.	`title`, `template`, `placeholders`, `in_navigation`, `soft_root`, `meta_description`.

For a page that exists in English and German, there is **one** `Page` row and **two** `PageContent` rows — one per language.

Add `djangocms-versioning` to the project and the same page now has *as many* `PageContent` rows per language as there are draft, published, unpublished, and archived versions.

Why split it

Three concerns all push in the same direction:

- **Translations.** A page is not really *in* a single language — it has a translation for each language. Putting language fields on the grouper would force one row per language, with everything that shouldn't vary by language (site, parent in the tree, apphook binding) duplicated and kept in sync by hand.
- **Versions.** A versioning package wants to keep many flavours of the same content around (current draft, last published, an archived earlier draft). If versions lived on the grouper, the page tree would have to grow a new node for every draft — broken.
- **Stable identity.** A page's URL, its position in the tree, and the apphook attached to it should not change when an editor saves a new translation or publishes a new version. The grouper is what *stays the same*.

The grouper / content split keeps each concern in the right place. The grouper is the *noun* (this page exists, here, in the tree). The content is the *adjective* (in this language, this version, with this title).

The concrete shape for pages

The two-part pattern is most visible on the page model:

```
Page                        (grouper)
 ├── site
 ├── parent / tree position
 ├── is_home
 ├── application_urls       ← the apphook binding
 │
 ├── PageContent rows       (content, per language × version)
 │    ├── language="en"
 │    │    ├── title="About us"
 │    │    ├── template="base.html"
 │    │    ├── placeholders → plugins
 │    │    └── in_navigation, soft_root, meta_description, ...
 │    │
 │    └── language="de"
 │         └── ...
 │
 └── PageUrl rows           (per-language URL, kept separate
                             from content for cache reasons)
```

The *Page <cms.models.pagemodel.Page>* docstring puts it plainly:

> A `Page` is an abstract entity. It does not have any content associated with it, nor does it provide any slugs to build a URL.

The pattern extends beyond pages

Other content objects follow the same shape:

- **Aliases** (from `djangocms-alias`): `Alias` is the grouper, `AliasContent` is the content. An alias survives across languages and versions in the same way a page does.

- **App-defined content objects.** Any add-on that needs versioning or translation on its own model is encouraged to use the grouper / content split. `djangocms-versioning` integrates with arbitrary grouper / content pairs through the `CMSAppExtension` contract.

The CMS provides scaffolding for the admin side of this pattern in *`GrouperModelAdmin`* — see *How to create an admin class for a grouper model* for how to use it on your own grouper / content pair.

Working with content in code

Two manager pairs come with every content model. They look the same on the outside; they answer very different questions.

`objects`
: Filters down to "content the **public** should see right now." What that means depends on which packages are installed — a versioning package narrows it to the currently published version in the current language, for example. Use this in any code path that serves a request.

`admin_manager`
: Returns **every** content row attached to the grouper, regardless of visibility, language, or version state. Use this in admin code, management commands, and tools that need to reason about the full history of a content object.

```
# In a view: only what's currently visible to the public.
PageContent.objects.filter(language="en", page=page)

# In an admin tool: every version, every language.
PageContent.admin_manager.filter(page=page)
```

Using `objects` in admin paths makes drafts and archived content invisible to editors. Using `admin_manager` in public paths leaks unpublished content. The split is intentional.

Implications and trade-offs

- **You always need to know which content row you mean.** "The page's title" is ambiguous — it's the title of *some* PageContent row. Most CMS APIs take a language argument (often defaulted from the request) to resolve this.
- **Foreign keys point at the grouper, not the content.** If your app's model needs to reference a page, the FK is to `Page`, not to `PageContent`. The reference survives translations and versions.
- **Custom admin work is more involved.** You are managing two models that should *feel* like one to the editor. *`GrouperModelAdmin`* exists to keep that ergonomics work out of your code.
- **Placeholders belong to the content, not the grouper.** Every language version of a page has its own placeholders, with its own plugins. That is by design: a German translation rarely consists of the same plugins in the same order as the English one.

Where to go next

- *How django CMS is composed* — how content objects compose with plugins and apphooks to form a site.
- *Publishing* — what "published" means for a content object, and what versioning packages add on top of the core split.
- *How to create an admin class for a grouper model* — building an admin for your own grouper / content pair.
- *How to share capabilities between apps* — declaring that your app's content models participate in CMS contracts (e.g. versioning).

Plugins

See also

- *How django CMS is composed* — where plugins fit alongside pages and apphooks, including the plugin-vs-apphook decision aid.
- *Plugins how-to guide*

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.

Why would you need to write a plugin?

A plugin is the most convenient way to integrate content from another Django application 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.

Components of a plugin

A django CMS plugin is fundamentally composed of three components, that correspond to Django's familiar Model-View-Template scheme:

What	Function	Subclass of
model (if required)	plugin instance configuration	*CMSPlugin*
view	display logic	*CMSPluginBase*
template	rendering	–

CMSPlugin

The plugin **model**, the sub-class of *cms.models.pluginmodel.CMSPlugin*, is 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.

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`
- `prepopulated_fields`
- `preserve_fields`
- `save_as`
- `save_on_top`
- `search_fields`
- `show_full_result_count`

- `view_on_site`

Beyond Python plugins: djangocms-frontend

Every plugin described so far requires a Python class — a `CMSPluginBase` subclass, and usually a model. djangocms-frontend offers two lighter-weight paths to a plugin, both translated into full django CMS plugins under the hood:

Template components. Write a Django template, place it in a `cms_components` directory inside one of your apps, and djangocms-frontend auto-detects it at startup. Fields are declared in the template itself — no Python file is needed.

Custom components. Write a Python class (subclassing `CMSFrontendComponent`) in a `cms_components.py` file, declare its fields as Django form field attributes, and register it with the `@components.register` decorator. This gives you full control over the add and change forms — fieldsets, custom validation, mixins — while still avoiding the boilerplate of a full `CMSPluginBase` subclass.

Both paths are framework-agnostic (the built-in components and mixins that ship with djangocms-frontend target Bootstrap 5, but you are not tied to it). Their trade-off is scope: template components cannot contain Python code at all, and custom components cannot add methods to the plugin or model class. When you need that full control, a `CMSPluginBase` subclass is the right tool.

For step-by-step tutorials and examples, see the djangocms-frontend documentation.

Application hooks ("apphooks")

See also

- *How django CMS is composed* — where apphooks fit alongside pages and plugins, including the plugin-vs-apphook decision aid.

An *Application Hook* (usually simply *apphook*) attaches a Django application's URL tree to a django CMS page.

Conceptually, an apphook turns a CMS page into a **mount point** for another application's URLs. Once attached, requests to that page (and to everything below it) are **routed into the hooked application**, rather than being served as normal CMS page content.

Think of it like including a URLconf in your project's `urls.py`, except that the *base path* is defined by content editors through the page tree.

```
/records/            -> handled by the attached application's URLs
/records/1984/       -> handled by the attached application's URLs
/records/...         -> handled by the attached application's URLs
```

In other words: the CMS page provides the **base path** (`/records`), and the application provides the **remainder of the URL space** (`/1984/`).

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

That would integrate the application into your *project*, but it would not be fully integrated into django CMS. For example:

- django CMS would not be aware of it and could allow editors to create a CMS page with the same `/records` slug, which could then 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 integration by making the CMS aware of that URL space.

What changes in request handling

Without an apphook, a request like `/records/1984/` is resolved by Django's URLconf first, and then (if it falls through to the CMS) by django CMS.

With an apphook in place, django CMS treats the apphooked page as a routing entry:

- It resolves the CMS page for the base path (`/records/`).
- If that page has an apphook attached (and is published), it delegates URL resolution for the remainder of the path (`1984/`) to the application's URLconf.

This delegation is why apphooks are sometimes described as "swallowing" the URL space below a page: the application becomes responsible for everything underneath that base path.

What changes with an apphook

When you attach an application to a CMS page:

- The CMS page's URL becomes the **base path** for the application (e.g. `/records`).
- The application takes over **all URLs below that base path** (e.g. `/records/1984`).
- django CMS can treat the application as part of the site's structure (for example, preventing unreachable slug conflicts and supporting menu integration).

In practice this means the CMS page behaves like a routing entry in your project's URLconf.

This also means the application can be served at a URL defined by content managers, and moved around in the site structure by moving the page.

An apphook only becomes active when an editor attaches it to a page (see *How to create apphooks* for the mechanics).

Important

Apphooks are part of the CMS *publishing* model.

- An apphook does not serve any public traffic until the page is **published**.
- This also implies that parent pages on the path need to be published.

Implications and trade-offs

Apphooks trade "page renders its own content" for "page becomes a mount point". That has a few practical consequences:

- **Slug conflicts become CMS-visible**. Because django CMS owns that base path, it can prevent editors from creating pages that would make the mounted application unreachable.
- **Menu integration becomes possible**. Mounted application pages can participate in the site's navigation (depending on the menu integration the application/apphook provides).
- **CMS workflow and permissions can apply**. At minimum, publishing controls whether the mount point is public; additional integration varies by application.

Warning

An apphook "swallows" all URLs below the apphooked page.

As a result, child pages under an apphooked page are not reliably reachable, because requests under that path are routed to the application.

Workarounds exist for advanced cases, but they require explicit routing back into the CMS; see the discussion in *How to create apphooks*.

Multiple apphooks per application

The same application can be attached multiple times, to different pages. Each attachment creates a different mount point (for example `/records` and `/results` could both route into the same application).

See *Attaching an application multiple times* for details.

In addition, an application can support multiple configurations so that each mount point can behave differently.

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 site editors to create multiple configuration *instances* that specify this behaviour.

Terminology (and how the pieces relate)

Apphooks involve three related concepts that are easy to blur together:

Concept	Defined by	Meaning
Apphook class	Developer	Code that connects a CMS page to an application's URLconf ("mount this application here").
Apphook configuration class	Developer	Describes which configuration options exist ("what can editors choose?").
Apphook configuration instance	Editor	Concrete configuration data selected for one mount point ("use cycling mode here").

The available configuration options are presented in an admin form and are determined by the application developer.

Important

An apphook (and therefore also an apphook configuration) serves no function until it is attached to a page. Also, until that page is **published**, the application will not be reachable for the public at that location.

An apphook also "swallows" all URLs below that of the page, handing them over to the attached application. If you add child pages under an apphooked page, django CMS cannot serve them reliably because requests under that path are routed to the application instead.

How a request becomes a page

django CMS builds on Django's request-to-response cycle — URL routing, middleware, template rendering, and the cache framework are all standard Django. This page explains how the CMS extends that cycle: which Django parts it hooks into, what it adds on top, and how the two layers interact when a request for a CMS-managed page arrives. Read it when you need to debug a page that is not showing, understand why a cache is not invalidating, or build a mental model of how all the pieces fit together.

If you have not yet read *How django CMS is composed* and *Content objects: the grouper / content pattern*, do that first. The concepts introduced there — pages, content objects, placeholders, plugins, apphooks, and the grouper / content split — are the vocabulary this page assumes.

The big picture

```
Browser:  GET /de/events/2026-summit/
                  |

1. Middleware preamble
   ApphookReload → CurrentPage → Language → Toolbar
                  |

2. URL resolution
   slug → PageUrl table → Page object  (or 404 / welcome)
                  |

3. Language determination
   URL prefix → session → cookie → Accept-Language → LANGUAGE_CODE
                  |

4. Content resolution
   Page + language → which PageContent row?  (published? fallback?)
                  |

5. Page cache gate
   Hit? → return cached response immediately
   Miss? → proceed to render, cache result for next time
                  |

6. Template selection
   PageContent.template → "base.html"
                  |

7. Placeholder rendering (per {% placeholder %} tag)
   Cache check → load plugin tree → render plugins
                  |

8. Menu building (if template uses {% show_menu %})
                  |

9. Response
```

1. The middleware preamble

Before the CMS view runs, Django's middleware stack fires in order.

ApphookReloadMiddleware
: Ensures the URL configuration is up-to-date when apphooks have been added or removed. A stale URLconf would cause `NoReverseMatch` errors in menu templates.

CurrentPageMiddleware
: Attaches `request.current_page` as a lazy object. The page is resolved once on first access and stored on the request as `request._current_page_cache` — an in-process attribute that prevents resolving the same page twice during a single request.

`LanguageCookieMiddleware` (optional — add it to `MIDDLEWARE` to enable)

: Persists the user's language preference in a cookie so that subsequent visits default to the same language.

ToolbarMiddleware
: Attaches the toolbar if the user is staff and the request is in edit mode (`?edit` or `?toolbar_on`). The toolbar's presence disables CMS-level caching for the request.

2. URL resolution

The request arrives at `cms.views.details()` via the CMS URLconf. The view calls `get_page_from_request()`, which matches the URL slug against the `PageUrl` table.

The `PageUrl` model stores one row per `(page, language)` pair, each with its own `path` field. This means the German URL `/de/ueber-uns/` and the English URL `/en/about-us/` are stored independently and resolve to the same `Page` through different `PageContent` rows.

If no page is found:

- The CMS checks whether the request falls inside an apphook's URL space (via `applications_page_check`). If it does, the apphook's parent page is treated as `request.current_page`.
- If the URL is `/` and no pages exist at all, the welcome screen is rendered.
- Otherwise, a 404 is raised.

A `PageUrl` lookup happens on most requests. To avoid hitting the database every time, the result is cached in Django's cache backend — keyed by `(page_lookup, language, site_id)`, with a duration set by *CMS_CACHE_DURATIONS* `['content']` (default 60 seconds). The cache is invalidated alongside the global page cache when a page's slug changes or the page is moved in the tree.

3. Language determination

By the time the CMS view runs, Django's `LocaleMiddleware` has already resolved the active language. The resolution chain, in order:

1. language prefix in the URL (`/de/` when using `i18n_patterns()`)
2. language stored in the user's session
3. language stored in a cookie (from `LanguageCookieMiddleware`)
4. browser's `Accept-Language` header
5. the project's `LANGUAGE_CODE`

The resolved language is available as `request.LANGUAGE_CODE`.

CMS_LANGUAGES overlays additional policy per language: `fallbacks`, `redirect_on_fallback`, `hide_untranslated`, and `public`. These are consulted in the next step, not here. Language *preference* (this step) and content *availability* (the next step) are separate concerns.

4. Content resolution

With the `Page` and the language known, the CMS needs to determine *which* `PageContent` row to serve.

A `Page` instance carries a `page_content_cache` dictionary — mapping `language` → `PageContent` — that is populated once per request by `_get_page_content_cache()`. It loads all available `PageContent` rows for the page in a single query, so subsequent lookups during the same request (including those from template tags and the toolbar) hit this dictionary rather than the database.

The lookup logic:

- **Direct hit.** A `PageContent` row exists in the requested language. Serve it.
- **Fallback hit.** No row in the requested language, but the language's `fallbacks` list (in *CMS_LANGUAGES*) names a language that does have one. If `redirect_on_fallback` is `True` (the default), the browser is redirected to the fallback language's URL. If `False`, the fallback's content is served under the original URL.
- **No fallback.** 404.

When `djangocms-versioning` is installed, `PageContent.objects` (the default manager) filters to the *published* row per language. `PageContent.admin_manager` returns every row and is used in admin code paths only. See *Publishing* for the full version-state model.

5. The page cache gate

Before any rendering begins, the CMS checks whether a cached copy of the entire page already exists. This is the most impactful cache in the system: a hit avoids template loading, placeholder rendering, plugin rendering, and menu building entirely.

The check runs only when *CMS_PAGE_CACHE* is `True` (default), the user is anonymous, the toolbar is not in edit mode, and no placeholder on the page uses the legacy `cache = False` flag (see **`Step 7`_**).

If the conditions are met, `get_page_cache(request)` queries Django's cache backend for a key composed from the site ID, language, and a hash of the request path. A hit returns `(content, headers, expires_datetime)` — the CMS reconstructs an `HttpResponse`, recalculates a `max-age` header from the stored expiry, and returns immediately. No further processing.

The page cache uses a versioning strategy: a global integer is stored under `CMS_PAGE_CACHE_VERSION_KEY`. Each cache write includes the current version. `invalidate_cms_page_cache()` increments the version — all previous entries become unreachable and expire naturally. The version key is re-written with a fresh timeout on every cache write, ensuring it always outlives the entries it protects.

The cache duration is the shortest of `CMS_CACHE_DURATIONS['content']` and the TTL returned by each rendered placeholder's *get_cache_expiration()*. If any placeholder returns `EXPIRE_NOW` (0), the page is not cached at all.

When the page cache is skipped — for authenticated users, toolbar edit mode, or plugins that opt out — the remaining cache layers (placeholder, menu) are also skipped for that request.

On a cache miss, the CMS proceeds to render the page (Steps 6–8). After rendering, `set_page_cache()` stores the result: it gathers all placeholders that were rendered, computes the effective TTL as the shortest across all placeholders, collects the union of Vary headers, and writes `(content, headers, expires_datetime)` to the cache. The absolute expiry timestamp is stored so that future reads can recalculate `max-age` without recomputing each placeholder's TTL.

6. Template selection

The `PageContent.template` field names the Django template to use (e.g. `"base.html"`). Template resolution follows Django's standard loader chain: the CMS template directory, then the project's template directories, then any additional loaders configured in `TEMPLATES`.

The template defines which placeholders exist through `{% placeholder "name" %}` tags. These are the rendering anchor points for the next step.

7. Placeholder rendering

For each `{% placeholder %}` tag in the template, the CMS resolves the corresponding *`Placeholder`* object — scoped to the current `PageContent` — and renders it.

Before loading plugins, the CMS checks the placeholder cache. This cache stores the fully rendered HTML for a single placeholder, keyed by placeholder ID, language, site, timezone, and any Vary headers the placeholder's plugins declare. It uses its own per-placeholder version integer — separate from the global page cache version — so invalidating one placeholder does not affect others. The check runs when *`CMS_PLACEHOLDER_CACHE`* is `True` (default), the toolbar is not in edit mode, and caching is enabled on the placeholder and in the template tag (both defaults).

A cache hit returns the pre-rendered HTML directly. On a miss, the CMS loads the plugin tree and renders it.

Plugin tree loading. Plugins form a tree through a self-referential foreign key: each `CMSPlugin` row has a `parent` field pointing to its container plugin (or `NULL` for plugins placed directly into a placeholder). The tree is assembled by querying children at each level, ordered by a `position` field.

The root plugins (those inserted directly into the placeholder) are determined by the template and slot; valid root plugin classes for a given `(template, slot)` are cached in-process in `PluginPool.root_plugin_cache`, avoiding recomputation on every render.

Plugin rendering. The tree is walked depth-first. For each plugin, *`CMSPluginBase.render()`* returns an HTML fragment. Context flows from parent to child through the *`CMS_PLUGIN_CONTEXT_PROCESSORS`* pipeline.

Plugin-level cache control. Each plugin class can influence the placeholder cache through two methods:

`get_cache_expiration(request, instance, placeholder)`
: Returns a TTL in seconds for this plugin instance. The placeholder uses the shortest TTL across all its plugins. Override this to vary cache duration by content type — a shorter TTL for a "latest news" plugin than for static footer content.

`get_vary_cache_on(request, instance, placeholder)`
: Returns a list of HTTP header names to vary on — for example `['User-Agent']` for a plugin that renders differently on mobile.

The legacy `cache = False` attribute on a `CMSPluginBase` subclass causes the plugin to return `EXPIRE_NOW`, which disables the page cache for any page containing that plugin. Prefer overriding `get_cache_expiration()` with a short TTL instead.

After rendering, the result is stored in the placeholder cache with a duration of `min(CMS_CACHE_DURATIONS['content'], placeholder_ttl)`. The cache is invalidated — by incrementing the per-placeholder version — whenever any plugin inside it is saved, moved, or deleted.

During rendering, plugins may also perform permission checks — for example, whether the current user can see a restricted plugin. These checks use the permission cache, which stores allowed page IDs per user and per action (`change_page`, `publish_page`, etc.) in Django's cache backend. It is invalidated when the user's groups change or when any `PagePermission` or `GlobalPagePermission` is saved or deleted. See *Permissions* for the full model.

8. Menu building

If the template uses `{% show_menu %}` or related tags, the menu system assembles the navigation tree.

The menu is cached independently of the page and its placeholders. A cached page response does *not* imply a cached menu — the menu is still built or retrieved on every request, because it depends on the page tree state, which can change independently of individual page content.

The menu cache is keyed by `(site_id, language)`, stored in Django's cache backend, with a duration set by *CMS_CACHE_DURATIONS* `['menus']` (default one hour). On a cache miss, the system runs menu generators (`CMSMenu` walks the page tree) and modifiers (soft root cutting, auth visibility filtering, level marking), then serializes the result. The cache is invalidated by `menu_pool.clear()` whenever a page is saved, moved, published, or deleted — selectively by site and language or globally.

Menu building and toolbar authorization also consult the permission cache (see **`Step 7`_**) to determine which pages are visible to the current user based on `hide_untranslated`, login-required, and view-restriction settings.

For the full menu system, see *How the menu system works*.

9. Response

The rendered page — whether from cache or freshly built — is returned as a Django `HttpResponse`. If the page cache layer wrote the response, the headers include `Cache-Control: max-age=...`, `Expires`, and `Vary`, set from the values computed during the cache write in Step 5.

The toolbar, if active, wraps each plugin in edit markers and injects structure-board data and admin URLs. This happens after rendering and does not affect the cache — edit-mode requests bypass all CMS-level caching and are never cached themselves.

Cache layers at a glance

Layer	Storage	Keyed by	Duration	Invalidated by
`request._current_page_cache`	Process memory	—	Request lifetime	—
`page.page_content_cache`	Process memory	language	Request lifetime	—
Page URL cache	Django cache	(page, lang, site)	`content` TTL	Version bump on slug/move
Page cache	Django cache	(site, lang, path hash)	`min(content, min ph TTL)`	`invalidate_cms_page_cache()`
Placeholder cache	Django cache	(ph, lang, site, tz, vary)	`min(content, ph TTL)`	`clear_placeholder_cache()`
Permission cache	Django cache	(user, action)	`permissions` TTL	Version bump on group/perm change
Menu cache	Django cache	(site, lang)	`menus` TTL	`menu_pool.clear()`
Plugin pool cache	Process memory	(template, slot)	Process lifetime	—

Where to go next

- *How django CMS is composed* — the three building blocks (content objects, plugins, apphooks) that the lifecycle orchestrates.
- *Content objects: the grouper / content pattern* — the grouper / content split that underpins content resolution (Step 4).

- *Publishing* — how version states affect which content rows are visible and what "published" means for caching.
- *Serving content in multiple languages* — the language-determination and fallback rules that drive Steps 3 and 4.
- *How the menu system works* — the generators and modifiers behind Step 8.
- *Permissions* — the permission model that the permission cache protects.
- *Configuring django CMS* — the authoritative reference for `CMS_PAGE_CACHE`, `CMS_PLACEHOLDER_CACHE`, `CMS_CACHE_DURATIONS`, and related settings.
- *Plugins* — the `CMSPluginBase` API, including `get_cache_expiration()` and `get_vary_cache_on()`.

Publishing

In django CMS, **"publishing" is not a core concept** — it is a contract that versioning packages plug into. This page explains what the core actually guarantees, what `djangocms-versioning` (the standard versioning package) adds on top, and why the model is structured this way.

If you have not yet read *Content objects: the grouper / content pattern*, do that first. The grouper / content split is the foundation everything below stands on.

Without a versioning package

The django CMS core has **no separate publish action**. There is no "draft" and there is no "published" — there is only the content row. When an editor saves a `PageContent` row, the change is immediately visible to anyone who can resolve a URL to that row.

In this mode:

- one `Page` can have many `PageContent` rows, but only **one per language**,
- `PageContent.objects` and `PageContent.admin_manager` return the same thing (there is nothing to hide),
- editing *is* publishing.

This is enough for development environments, single-author sites, or projects whose editorial workflow lives outside the CMS (e.g. content prepared in another tool and imported).

It is rarely enough for a production site with editors.

With `djangocms-versioning`

djangocms-versioning is the standard versioning package endorsed by the django CMS Association. The quickstart install includes it. It plugs into the CMS through the `CMSAppExtension` contract and changes three things about how content objects behave:

1. **Many content rows per language are now allowed.** Each new draft creates a new `PageContent` row. The grouper / content split was already there; versioning is what fills it with more than one row per language.
2. **Each content row gets a state.** *Draft*, *published*, *unpublished*, or *archived*. The state is stored on a separate `Version` model that points at the content row.
3. **The default manager filters by state.** `PageContent.objects` now means "the *published* row for each language." `PageContent.admin_manager` is the escape hatch that still returns every row.

Editing is no longer publishing. The "Publish" button promotes the current draft to published; the previous published row becomes unpublished; the new draft, when an editor next edits, becomes the new draft row.

Version states

When `djangocms-versioning` is installed, each content row carries one of four states.

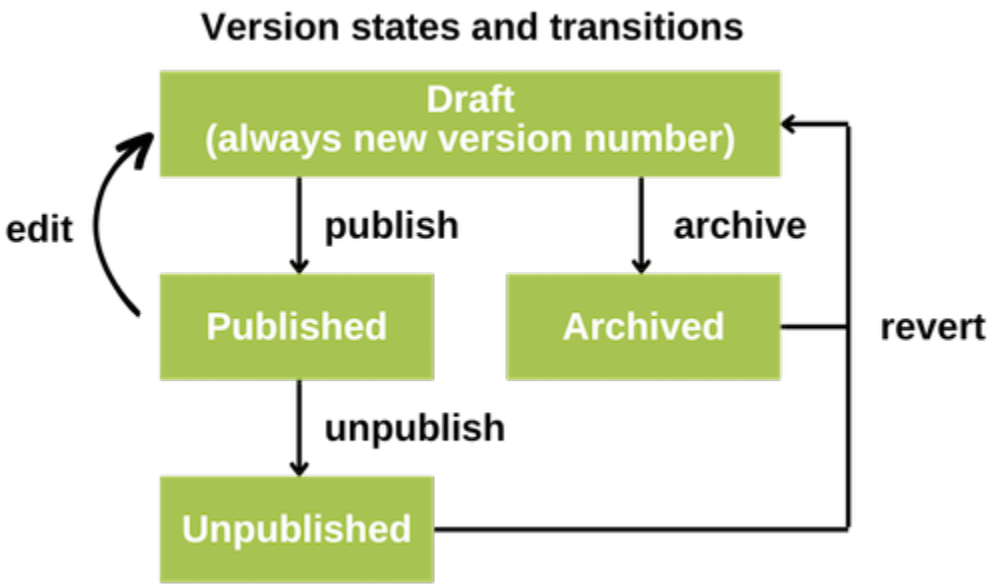
Draft
: The version currently being edited. **Only one draft per language** at any time. Drafts are not visible to the public.

Published
: The version currently visible on the site. **Only one published version per language** at any time. Published rows cannot be edited in place — editing creates a new draft based on the published row.

Unpublished
: A row that was previously published and has been taken offline. Many unpublished rows can coexist, preserving the history of what was once live.

Archived
: A row that was never published. Archived rows preserve work that may be useful later and can be reverted to draft.

Each new draft increments a version number, giving a complete history of changes over all historically created versions including archived, and unpublished ones.



A page is publicly reachable when its current-language `PageContent` row is in the *Published* state. Whether the page's parents are published does not affect its own reachability — pages stand on their own URLs.

Apphooks and publishing

An apphook attached to a page **inherits the page's publishing state**:

- the apphook is reachable only when the page is published,
- unpublishing the page takes the apphook offline as well.

This is consistent with how content objects compose with apphooks (*How django CMS is composed*): an apphook is a binding *on* the page, so it shares the page's visibility.

The scope is wider than pages

`djangocms-versioning` does not version *only* pages. The contract applies to any grouper / content pair an app registers. The most common second example is djangocms-alias — aliases get the same draft / published / unpublished / archived states as pages, with the same editorial flow.

If your own app has a content model that should support drafts and versioning, build it as a grouper / content pair (see *Content objects: the grouper / content pattern*) and register it via the `CMSAppExtension` contract.

Working with versions in code

In any code path that **serves a request**, use the default manager:

```
PageContent.objects.filter(language="en")
# ← only the published row per language
```

In **admin** code, management commands, or anywhere you legitimately need access to every version, use the admin manager:

```
PageContent.admin_manager.filter(page=my_page)
# ← every row, regardless of state or language
```

To fetch a specific version state explicitly, query the `Version` model directly:

```
from djangocms_versioning.constants import DRAFT
from djangocms_versioning.models import Version

draft = Version.objects.get(
    content__page=my_page,
    content__language="en",
    state=DRAFT,
).content
```

To iterate "the current row for each language" — draft if one exists, otherwise published — use `current_content()`:

```
qs = PageContent.admin_manager.filter(page=my_page).current_content()
```

See the djangocms-versioning documentation for the rest of the API.

Alternative versioning packages

The CMS uses a contract-based approach (`CMSAppExtension`) rather than hard-wiring `djangocms-versioning` into the core. That means alternative implementations are possible.

djangocms-no-versioning provides a minimal publish / unpublish toggle without keeping a full version history. Useful when basic visibility control is enough and the storage and UI overhead of full versioning is not justified.

The contract also means a project that wants moderation workflows on top of versioning can install djangocms-moderation, which adds approval chains before a draft can become published.

Considerations when choosing or switching

The versioning package is project-wide. It applies to every grouper / content pair that opts into the contract — pages, aliases, and any app-defined content models — not just to pages.

Switching versioning packages on an existing project is not a trivial migration. Different packages store version metadata in different models. The standard path is to uninstall the current package (leaving an unversioned CMS), then

install the replacement and let it create its own data structures. Existing draft / archived state typically does **not** carry across; only the most-recent content row per language survives. Plan for this.

Where to go next

- *Content objects: the grouper / content pattern* — the grouper / content pattern this page relies on.
- *How django CMS is composed* — how content objects, plugins, and apphooks combine to form a site.
- *Permissions* — how publishing permissions are granted.
- *How to share capabilities between apps* — declaring your own content models as participants in the versioning contract.

Serving content in multiple languages

django CMS treats multilingual content as a first-class concern, not as a feature bolted on top of single-language pages. This page describes the underlying model, the configuration surface that controls it, and the decisions every multilingual project has to make.

For step-by-step setup, see *How to serve multiple languages*. For the authoritative list of settings, see *Configuring django CMS*.

Content per language

Every content object follows the grouper / content split described in *Content objects: the grouper / content pattern*. The grouper (`Page`, `Alias`) is language-agnostic — it owns the identity, the tree position, the apphook binding. The content row (`PageContent`, `AliasContent`) is **per language**.

That means:

- a page that exists in English and German is **one** `Page` with **two** `PageContent` rows,
- each translation has its **own placeholders** and its own plugins, *and* its own `title`, `meta_description`, `template` choice, and `in_navigation` setting,
- slugs are also per language — they live in a separate `PageUrl` row keyed by `(page, language)`, which is why the German URL `/de/ueber-uns/` and the English URL `/en/about-us/` can be completely independent words.

Translating a page is therefore not a content edit — it is **creating a new PageContent row** for that page in the target language. The grouper-side fields (apphook binding, tree position, login required) are not duplicated.

`CMS_LANGUAGES` and Django's `LANGUAGES`

Django already has `LANGUAGES` and `LANGUAGE_CODE`. django CMS adds its own *`CMS_LANGUAGES`* setting on top. They do different jobs:

- `LANGUAGES` is the set of languages your *project* knows about — Django uses it for translation files, form labels, URL routing.
- *`CMS_LANGUAGES`* is how each of those languages should behave **as CMS content**: which sites it is offered on, what to do when a page is not translated into it, whether to show pages in it to anonymous visitors, and what to fall back to.

The per-language options that matter most are:

`fallbacks`
: Ordered list of language codes to try if a requested `PageContent` row does not exist in this language.

`redirect_on_fallback`
: When a fallback is served, whether to redirect the browser to the fallback's own URL (default `True`) or to keep the URL of the originally requested language (and serve the fallback content under it).

`hide_untranslated`
: Whether pages without a `PageContent` row in this language are hidden from menus. Default `True`.

`public`
: Whether this language is offered to anonymous visitors. Default `True`. Setting this to `False` is useful for staging a translation before it goes live.

The setting can be configured per site (for multi-site projects), with a `default` block providing fall-through values.

How django CMS determines which language to serve

Serving a multilingual page is two questions, asked in order:

1. **Which language does the visitor want?** (language *preference*)
2. **Does the requested page exist in that language?** (content *availability*)

Each question has its own resolution rules. Conflating them is a common source of "why is it serving English?" bugs.

Language preference

django CMS uses Django's standard language-discovery chain, in this order:

- the language code prefix in the URL (e.g. `/de/` when using `i18n_patterns()`),
- the language stored in the user's session,
- the language stored in a cookie from a previous visit (set by `cms.middleware.language.LanguageCookieMiddleware` — not on by default),
- the language requested by the browser in the `Accept-Language` header,
- the project's `LANGUAGE_CODE` as a final fallback.

The first match wins. More detail is in the Django docs at How Django discovers language preference.

Content availability

Once the preferred language is known, the CMS looks up the `PageContent` row for the requested page in that language.

- **Direct hit.** A `PageContent` row exists in the requested language — serve it.
- **Fallback hit.** No row in the requested language, but `fallbacks` lists another language that *does* have one. What happens next depends on `redirect_on_fallback`:
  - `True` (default): redirect the browser to the URL of the fallback language. The address bar now shows the fallback's URL prefix.
  - `False`: serve the fallback's content under the originally requested URL. The address bar keeps the requested language prefix but the content is in the fallback language.
- **No fallback.** No row exists in the requested language and no `fallbacks` entry produces a hit either — the CMS returns 404.

These three behaviours cover most real cases. Be deliberate about which one each language gets; "serve English under the German URL" is rarely what a German-speaking visitor wants.

URL strategies

A multilingual django CMS site picks one of three URL shapes. The choice affects deployment, SEO, and the editor experience.

Prefixed URLs (`/en/`, `/de/`)
: The most common. Wrap your CMS URL patterns in `i18n_patterns()`. Django strips the prefix and activates the matching language; the CMS resolves the page in that language. One domain, all languages.

Per-domain (`example.com` and `example.de`)
: One Django `Site` per language. `SITE_ID` selects the active site per request (usually via host-header middleware); each site's *`CMS_LANGUAGES`* block defines what languages it serves. URLs do not carry a language prefix. Best when SEO or branding requires distinct domains.

No URL marker
: No prefix, no per-domain split. The language comes entirely from the cookie, session, or `Accept-Language` header. Rare in practice — most visitors and search engines expect a stable URL per language.

Per-language menus and visibility

Because content is per-language, navigation is too. The CMS menu shows a page in language X if there is a `PageContent` row for it in language X *and* that row's `in_navigation` flag is set.

If a page has no row in the requested language:

- with `hide_untranslated = True` (default), the page is omitted from menus in that language,
- with `hide_untranslated = False`, the page is shown in menus even in languages it has not been translated into; clicking follows the fallback rules above.

The `public` per-language flag is independent: a language marked `public = False` is hidden from anonymous visitors entirely (useful for staging translations before launch), regardless of which pages are translated into it.

Where to go next

- *Content objects: the grouper / content pattern* — the grouper / content split that makes per-language content possible.
- *How django CMS is composed* — how content objects, plugins, and apphooks combine; placeholders being per-language is part of this story.
- *How to serve multiple languages* — practical steps to enable multilingual support in a project (settings, middleware, URL patterns).
- *Configuring django CMS* — the authoritative reference for *`CMS_LANGUAGES`* and related settings.

Permissions

django CMS sits on top of Django's standard auth system and adds a second, optional layer of **per-page permissions** for projects that need finer control. This page explains how those two layers fit together, the roles they are designed to support, and the trade-offs that come with each.

Permission modes

Permissions operate in two different modes, depending on the *`CMS_PERMISSION`* setting.

- Simple permissions mode (`CMS_PERMISSION = False`): only the standard Django Users and Groups permissions will apply. This is the default.

- Page permissions mode (`CMS_PERMISSION = True`): as well as standard Django permissions, django CMS provides row-level permissions on pages, allowing you to control the access of users to different sections of a site, and sites within a multi-site project.

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. These apply equally in Simple permissions mode and Page permissions mode.

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

- `django CMS | cms plugin`
- `django CMS | page`
- `django CMS | placeholder`
- `django CMS | placeholder reference`

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

In addition to the core package permissions, an editor will likely need the following permissions from 3rd-party packages:

- djangocms-alias
  - `django CMS Alias | alias`
  - `django CMS Alias | alias content`
  - `django CMS Alias | category`
- djangocms-frontend
  - `django CMS Frontend | UI item`
  - After adding these permissions, save and use the `python manage.py frontend sync_permissions` command as documented in djangocms-frontend's documentation
- djangocms-text
  - `django CMS Rich Text | text`
- djangocms-versioning
  - `django CMS Versioning | alias content version`
  - `django CMS Versioning | page content version`
  - `django CMS Versioning | version`

Typically when adding other 3rd party packages or custom plugins you may need to add additional permissions to enable their features. Sometimes documentation for such needed permissions may be missing, in that case you can compare the list of available permissions with the package enabled and disabled on your site.

See use-permissions-on-groups below on applying permissions to groups rather than users.

Permissions in Page permissions mode

In Page permissions mode, you also need to give users permission to the right pages and sub-sites.

Global and per-page permissions

Roles, not just users

A team using a CMS usually has more than one kind of user. django CMS is shaped around four loose roles, even though Django itself only knows about users and groups:

Author
: Creates and edits draft content. Cannot publish. Often the largest group on a content-heavy site.

Editor
: Reviews, edits, and publishes. The role that needs publish permission; otherwise looks like an author with more rights.

Designer
: Owns templates, styles, and the layout vocabulary. Works in code, not in the toolbar. Permission-wise this is usually a developer with deployment access — not a CMS user role at all.

Site administrator
: Manages users, groups, permissions, deployment, and project configuration. Often a Django superuser.

The permission system below is the mechanism that lets you map these roles onto users and groups. The system does not care what you call your groups; it only enforces what each group is allowed to do.

Two layers, two modes

There are two layers of permissions in any django CMS project:

1. **Django auth permissions** (always on). The standard per-model add / change / delete permissions Django gives to every model. CMS models — `Page`, `PageContent`, `Placeholder`, `CMSPlugin`, and so on — participate in this system like any other Django model. Configured in the Django admin under *Authentication and Authorization*.
2. **CMS per-page permissions** (off by default). Row-level permissions on individual pages or page subtrees: "this group can edit pages under `/legal/`, but not under `/marketing/`". Configured per project by the *CMS_PERMISSION* setting.

The setting controls which model the second layer follows:

- `CMS_PERMISSION = False` — only Django auth applies. Whoever can edit pages can edit *all* pages. Simple, predictable, sufficient for small teams and single-domain sites.
- `CMS_PERMISSION = True` — Django auth still applies, *and* on top of it the CMS checks per-page permissions for each request. Needed when different groups should own different parts of the page tree, or when a multi-site project needs editors scoped to one site.

Turning `CMS_PERMISSION` on is not free: every page operation now involves additional database lookups, and the admin grows several new forms that a small team may find more confusing than helpful. Switch it on when you need it, not by default.

Two dimensions of permission

Whichever mode is on, permissions divide along **two independent dimensions**:

- **What** the user is allowed to do — add, change, delete, publish, change advanced settings, move pages in the tree.
- **Where** they are allowed to do it — globally (all pages on a site), or only on a specific page subtree.

A *Basic editor* group might have "can change" on the *what* axis and "all pages on site X" on the *where* axis. A *Legal team* group might have the same *what* permissions but be restricted to the `/legal/` subtree on the *where* axis.

The *where* dimension only exists when `CMS_PERMISSION = True`. In the simpler mode, every CMS permission applies to every page.

The two models that implement the *where* dimension are:

- `GlobalPagePermission` — applies to all pages of one or more sites.
- `PagePermission` — applies to a specific page, optionally cascading to its descendants.

Permissions for plugins, not just pages

A common surprise: granting a user "can change" on a page does *not* let them add or edit the plugins inside that page. Plugins are their own model with their own permissions.

The split is intentional. Plugins are reusable components defined by add-on packages; their permissions are managed alongside the package that defines them, not alongside the page. To grant an editor the right to add and edit plugins on a page they can already edit, give their group the standard Django auth permissions on the relevant plugin models (`djangocms_text.text`, `djangocms_frontend.uiitem`, your custom plugin model, and so on).

If this feels like permission sprawl, you are not alone. The *How to share capabilities between apps* mechanism and convenience admin actions exist partly to reduce the per-package plumbing.

Publishing and the versioning package

"Can publish" is the most-asked-about CMS permission. It is also the one most affected by which versioning package is installed.

Without a versioning package, **there is no publish action** to permit. Saving a `PageContent` row is what makes it visible. See *Publishing*.

With `djangocms-versioning` installed, **publish becomes a distinct step** with its own permission, separate from "change". A common group design is:

- *Authors* — can add and change page content; cannot publish.
- *Editors* — same as authors, plus can publish.

The "Can publish" permission is granted on the relevant content version models exposed by `djangocms-versioning` (e.g. `django CMS Versioning | page content version`). Different versioning packages may model this differently; check the package's own documentation for the exact permission names.

If your project uses *How to share capabilities between apps* to register custom content models, those models participate in the same versioning contract and inherit the same publish permission model.

View restrictions vs edit permissions

A separate concern, often confused with edit permissions: who is allowed to *see* a published page.

- **Login-required pages.** Available without `CMS_PERMISSION`. A page can be marked as requiring login; anonymous visitors are redirected to log in.
- **View restrictions per group.** Available only with `CMS_PERMISSION = True`. A page can be restricted so that only members of specific groups can see it. Useful for intranet sections, customer portals, or pre-launch staging pages.
- **Menu visibility.** Independent of view restrictions: a page can be hidden from menus while remaining reachable by URL, or shown only to anonymous (logged-out) visitors, or only to authenticated ones.

These three controls are layered. A page can be in the menu for logged-in users (menu visibility), require login to view at all (login-required), and be further restricted to one group (view restriction).

These are front-end controls only. View restrictions, login-required, and menu visibility all govern who can *see a published page on the public site*. They do **not** hide a page inside the admin. The page tree in the admin — and the page-link autocomplete used by the smart-link field, which mirrors it — list *every* page on a site to any staff user who can edit at least one page on that site. Such a user can therefore see the titles, paths, and URLs of restricted,

login-required, and draft pages, even ones they cannot view on the front end or edit. Per-page edit permissions gate the *actions* offered on each node (edit, move, delete), not whether the node is listed.

In other words, a page title or path is not a secret from your staff editors. If a page's *existence* must be hidden from some staff users, the page tree is the wrong tool: keep that content on a separate site (see *CMS_PERMISSION* and multi-site setups) or outside the CMS, rather than relying on a view restriction to conceal it in the admin.

Delegated user management

With `CMS_PERMISSION = True` a non-superuser can be given the right to manage *other* users — the "Users" and "User groups" entries in the admin become available to anyone who has the `change` permission on the CMS user/group models and a page-permission level of their own. These users are **page-user managers**. They are not superusers, yet inside their own corner of the system they act with superuser-like authority.

The mental model is deliberate: **a page-user manager is a superuser for their subordinate users only.** A user is "subordinate" when the manager created them, or when they sit at the same or a lower level in the page tree the manager controls. Within that subordinate set, the manager can do almost everything a superuser could do to those accounts:

- create new staff users (new page-users are made staff automatically);
- grant and revoke any permission or group the manager *themselves* holds — they cannot hand out rights they do not have;
- edit account status fields, including `is_staff` (admin-login capability) and `is_active` (whether the account may log in at all).

The single boundary a manager cannot cross is **superuser status**: `is_superuser` is read-only for non-superusers, so a manager can never promote a subordinate (or themselves) to full superuser.

A manager can reverse a setting a superuser made. This follows directly from the model and is worth stating plainly. If a superuser disables a subordinate account (`is_active = False`) or removes its staff flag (`is_staff = False`), a page-user manager with that user in their subordinate set can switch it back on. The manager's authority over a subordinate is not subordinate to the superuser's earlier edit; it is the *same* authority over that account, minus the ability to grant superuser. If you need a deactivation or a demotion to be permanent against a manager, the user must be moved out of that manager's subordinate set — for example by deleting the account, or by re-parenting it above the manager's page-tree level — rather than relying on the status flag alone.

This is intentional delegation, not a gap: the whole point of a page-user manager is to off-load routine account administration from the superuser. Hand the role only to people you would trust with the accounts it covers.

Strategy

A few guidelines that hold regardless of which mode you are in:

Apply permissions to Groups, not Users. Per-user permissions drift quickly. After a year, no one will remember why a specific user has a specific permission. Group-based grants survive staff changes and are auditable.

Compose Groups by responsibility, not by person. A *Basic editor* group, a *Lead editor* group, a *Blog editor* group, a *Legal* group. Users land in one or more of these based on what they do. Avoid groups named after individuals or departments ("Marketing") unless the departmental boundary is also the permission boundary.

Start in simple mode. Leave `CMS_PERMISSION = False` until you have a concrete reason to switch. Most teams that turn it on at the start later wish they had not — the additional admin surface is real, and re-engineering away from it is harder than adopting it later.

Permissions are not a substitute for trust. Anyone with the "change permissions" right can grant themselves more rights. The boundary that matters most in practice is who gets superuser; tighten that first.

Where to go next

- *Content objects: the grouper / content pattern* — pages, page content, and the grouper / content split that the permission system protects.
- *Publishing* — how publishing depends on the versioning package, and what that means for the "Can publish" permission.
- *Permissions* — the authoritative list of CMS permission models and fields.
- *Configuring django CMS* — the `CMS_PERMISSION` setting and related options.

How the menu system works

Basic concepts

Soft Roots

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

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

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

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

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

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

which is frankly overwhelming.

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

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

Registration

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

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

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

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

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

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

Generators and Modifiers

Menu generators and modifiers are classes.

Generators

To add nodes to a menu a generator is required.

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

These 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.

The ShowMenu.get_context() method

When the Django template engine encounters the {% show_menu %} template tag, it calls the *get_context()* of the ShowMenu class. get_context():

- calls *menus.menu_pool.MenuPool.get_nodes()* (see *The MenuPool.get_nodes() method* below)
- cuts any nodes other than its descendants (if a root_id has been provided)
- calls *menus.templatetags.menu_tags.cut_levels()* to remove unwanted levels
- calls *menus.menu_pool.MenuPool.apply_modifiers()* with post_cut = True
- return the nodes to the context in the variable children

The MenuPool.get_nodes() method

menus.menu_pool.MenuPool.get_nodes() calls three other methods of MenuPool in turn:

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

Checks every application's `cms_menus.py`, and registers:

- 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`); for each:
    - calls 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 (see *Menu Modifiers* below) in `self.modifiers` calling each one's *modify()* with `post_cut=False`.

Menu Modifiers

Each `Modifier` manipulates menu nodes and their attributes.

The default Modifiers, in the order they are called, are:

- *cms.cms_menus.NavExtender*
- *cms.cms_menus.SoftRootCutter*

  If `post_cut` is `True`, removes all nodes below the appropriate soft root; otherwise, returns immediately.
- *menus.modifiers.AuthVisibility*

  Removes nodes that require authorization to see
- *menus.modifiers.Level*

  Loops over all nodes; for each one that is a root node (`level == 0`) passes it to:
  - *mark_levels()* recurses over a node's descendants marking their levels

Frontend integration

Generally speaking, django CMS is wholly frontend-agnostic. It doesn't care what your site's frontend is built on or uses.

The exception to this is when editing your site, as the django CMS toolbar and editing controls use their own frontend code, and this can sometimes affect or be affected by your site's code.

The content reloading introduced in django CMS 3.5 for plugin operations (when moving/adding/deleting etc) pull markup changes from the server. This may require a JS widget to be reinitialised, or additional CSS to be loaded, depending on your own frontend set-up.

For example, if using Less.js, you may notice that content loads without expected CSS after plugin saves.

In such a case, you can use the `cms-content-refresh` event to take care of that, by adding something like:

```
{% if request.toolbar and request.toolbar.edit_mode_active %}
    <script>
    CMS.$(window).on('cms-content-refresh', function () {
         less.refresh();
    });
    </script>
{% endif %}
```

after the toolbar JavaScript.

Some commonly-used plugins

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.

All packages officially endorsed by the django CMS Association are listed in the django-cms/djangocms-ecosystem repository.

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.

You will always find a complete list of the officially endorsed core addons in the django CMS ecosystem github repository.

Here's a brief overview of the current status of the core addons:

Package	Description	Status	Supported Versions
djangocms-alias	Universal alias plugin for all contents that needs to be repeated within a project	production	4.1, 5.0, 5.1
djangocms-attribute-fields	An opinionated implementation of JSONField for arbitrary HTML element attributes	production	3.11, 4.1, 5.0, 5.1
djangocms-audio	django CMS Audio is a set of plugins for django CMS. That allow you to publish audio files on your site	production	3.11, 4.1, 5.0, 5.1
djangocms-file	django CMS File is a set of plugins for django CMS that allow you to add files for download to your site.	production	3.11, 4.1, 5.0, 5.1
djangocms-form-builder	Flexible HTML forms for your django CMS projects	alpha	3.11, 4.1, 5.0, 5.1
djangocms-frontend	django CMS Frontend facilitates the easy creation of reusable frontend components. It supports any CSS framework. For immediate use, it includes a comprehensive set of Bootstrap 5 components and templates	production	3.11, 4.1, 5.0, 5.1
djangocms-link	Universal link plugin - allows intuitive in-text linking to all known linkable objects (starting with CMS pages). Provides a django CMS link field and widget for developers to use in custom content elements. Compatible with (and enabled in) djangocms-text	production	3.11, 4.1, 5.0, 5.1
djangocms-moderation	Moderation workflows for django CMS and django CMS versioning	production	4.1, 5.0, 5.1
djangocms-picture	django CMS Picture is a plugin for django CMS that allows you to add images on your site	production	3.11, 4.1, 5.0, 5.1
djangocms-rest	djangocms-rest enables frontend projects to consume django CMS content through a browsable read-only, REST/JSON API It is based on the django rest framework (DRF).	production	4.1, 5.0, 5.1
djangocms-snippet	django CMS Snippet provides a plugin for django CMS to inject HTML, CSS or JavaScript snippets into your website	production	3.11, 4.1, 5.0, 5.1
djangocms-stories	django CMS blog application - Support for multilingual posts, placeholders, social network meta tags and configurable apphooks	beta	4.1, 5.0, 5.1
djangocms-text	Text Plugin for django CMS using Tiptap (or any other text editor of your choice)	beta	3.11, 4.1, 5.0, 5.1
djangocms-text-ckeditor5	Fully-fledged CKEditor5 frontend for djangocms-text.	beta	3.11, 4.1, 5.0, 5.1
djangocms-transfer	Import/Export (JSON) for page contents	production	3.11, 4.1, 5.0,

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

Third-party Addons

These are popular addons maintained by a third party.

Package	Description	Status	Supported Versions
djangocms-katex	Provides a django CMS plugin to render mathematical and chemical formulae written in the LaTeX languageusing.	production	3.11, 4.1, 5.0, 5.1
djangocms-markdown	A markdown content plugin and model field for django CMS. Write content in Markdown using an integrated editor and have it rendered as HTML on your site.	production	3.11, 4.1, 5.0, 5.1
djangocms-simple-admin-style	Styles the Django admin using the Django CMS theme with minimal additional CSS.	production	3.11, 4.1, 5.0, 5.1
djangocms-timed-publishing	django CMS versioning add on, that allows for setting start and end time for an object's visibility when publishing	production	4.1, 5.0, 5.1

Deprecated Addons

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

Package	Description	Status	Supported Versions
djangocms-admin-style	django CMS Admin Style is a Django Theme tailored to the needs of django CMS.	production	4.1, 5.0, 5.1
djangocms-bootstrap4	django CMS Bootstrap 4 is a plugin bundle for django CMS providing several components from the popular Bootstrap 4 framework. djangocms-frontend offers an automatic migration	production	3.11, 4.1, 5.0
djangocms-googlemap	django CMS Google Map is a set of plugins for django CMS that allow you to implement Google Map into your website.	production	3.11
djangocms-text-ckeditor	Text Plugin for django CMS using CKEditor 4 (now sunset)	production	3.11, 4.1, 5.0

Also, all Aldryn plugins are deprecated and archived.

Search and django CMS

See also

- *Version States*
- djangocms-versioning
- django-haystack

If you already have a background in django CMS and are coming from the 3.x days, you may be familiar with the package `aldryn-search` which was the recommended way for setting up search inside your django CMS project. However, with the deprecation of it and the new versioning primitives in django CMS 4.x being incompatible with it, you may wonder where to go from here.

To keep it simple, you can install an external package that handles implementing the search index for you. There are multiple options, such as

- djangocms-aldryn-search
- djangocms-haystack
- … and others

The idea is that we implement the search index just as we would for normal Django models, e.g. use an external library to handle the indexing (`django-haystack` in the above examples) and just implement the logic that builds the querysets and filters the languages depending on what index is currently active. `django-haystack` exposes many helpful utilities for interacting with the indexes and return the appropriate results for indexing.

To get an idea on how this works, feel free to look at the code of the above projects. A very simple index could look something like this:

```
from cms.models import PageContent
from django.db import models
from haystack import indexes


class PageContentIndex(indexes.SearchIndex, indexes.Indexable):
    text = indexes.CharField(document=True, use_template=False)
    title = indexes.CharField(indexed=False, stored=True)
    url = indexes.CharField(indexed=False, stored=True)

    def get_model(self):
        return PageContent

    def index_queryset(self, using=None) -> models.QuerySet:
        return self.get_model().objects.filter(language=using)

    def prepare(self, instance: PageContent) -> dict:
      self.prepared_data = super().prepare(instance)
      self.prepared_data["url"] = instance.page.get_absolute_url()
      self.prepared_data["title"] = instance.title
      self.prepared_data["text"] = (
        self.prepared_data["title"] + (instance.meta_description or "")
      )
      return self.prepared_data
```

Hint

Your index should be placed inside a `search_indexes.py` in one of your `INSTALLED_APPS`!

The above snippet uses the standard `text` field that is recommended by `django-haystack` to store all the indexable content. There is also a separate field for the title because you may want to display it as a heading in your search result, and a field for the URL so you can link to the pages.

The indexed content here is *not* using a template (which is one of the options to compose fields into an indexable field) but rather concatenates it manually using the `prepare` method which gets called by `django-haystack` to prepare data before the indexing starts.

As you can see in the `index_queryset` method, we only return those `PageContent` instances that are `published` and have a language matching the currently used Haystack connection key.

The `PageContent` then get passed into the `prepare` method one by one, so we can use the `instance.page` attribute to get the page meta description and use it as indexable text as well as the title of the current `PageContent` version.

Finally, you need to set your `HAYSTACK_CONNECTIONS` to contain a default key as well as **any language that you want to be indexed** as additional keys. You could also use different backends for your languages as well, this is up to you and how you want to configure your haystack installation. An example could look somewhat like this:

```
...

HAYSTACK_CONNECTIONS = {
  'default': {
        "ENGINE": "haystack.backends.whoosh_backend.WhooshEngine",
        "PATH": os.path.join(ROOT_DIR, "search_index", "whoosh_index_default"),
  },
  "en": {
      "ENGINE": "haystack.backends.whoosh_backend.WhooshEngine",
      "PATH": os.path.join(ROOT_DIR, "search_index", "whoosh_index_en"),
  },
  "de": {
      "ENGINE": "haystack.backends.whoosh_backend.WhooshEngine",
      "PATH": os.path.join(ROOT_DIR, "search_index", "whoosh_index_de"),
  }
}

...
```

Hint

This should be configured in your projects `settings.py`!

Now run `python manage.py rebuild_index` to start building your index. Depending on what backend you chose you should now see your index at the configured location.

You can inspect your index using a `SearchQuerySet`:

```
from haystack.query import SearchQuerySet

qs = SearchQuerySet(using="<your haystack connection alias / language key>")
```

(continues on next page)

(continued from previous page)

```
for result in qs.all():
  print(result.text)
```

Now it's up to you to add custom indexes to your own models, build views for your `SearchQuerySet` to implement a search form and much more.

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.

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.

Color schemes (light/dark) with django CMS

Important

These notes about the color scheme 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.

The admin interfaces will only reflect the described behavior if the package `djangocms-simple-admin-style` is installed. If it is not installed, the admin interface is managed by your underlying Django installation, which usually uses the browser's color scheme.

Setting the color scheme

Django CMS' default color scheme is `"light"`. To change the color scheme use the `CMS_COLOR_SCHEME` setting in your project's `setting.py`:

`CMS_COLOR_SCHEME = "light"`
: This is the default appearance and shows the interface with dark text on a white background.

`CMS_COLOR_SCHEME = "dark"`
: This so-called dark mode shows light text on a dark background.

`CMS_COLOR_SCHEME = "auto"`
: The auto mode chooses either light or dark color scheme based on the browser or operating system setting of the user.

Hint

If you plan to fix the color scheme to either light or dark, add a corresponding `data-theme` attribute to the `html` tag in your base template, e.g.

```
<html data-theme="light">
```

This will pin the color scheme early when loading pages and avoid potential flickering if the browser preference differs from the `CMS_COLOR_SCHEME` setting.

Changed in version 3.11.4: Before version 3.11.4 the color scheme was set by `data-color-scheme`. Since version 3.11.4 django CMS uses `data-theme` just as Django since version 4.2.

Important

Not all plugin admin interfaces might support a dark color scheme, especially if plugin forms contain custom widgets.

Toggle button for the color scheme

The setting `CMS_COLOR_SCHEME_TOGGLE` in the project's `settings.py` determines if a toggle icon (sun/moon/auto) is shown in the toolbar. It allows a user to switch their color scheme for their session.

By default, `CMS_COLOR_SCHEME_TOGGLE` is set to `True`.

Make your own admin CSS color scheme aware

Plugin forms or any admin forms use Django's admin app which itself supports light and dark color schemes. djangocms-simple-admin-style introduces django CMS' color scheme to the admin app. Just as Django does, djangocms-simple-admin-style defines CSS variables for frequent colors.

We recommend writing at least your reusable apps in a way which allows them to respect the color scheme with djangocms-simple-admin-style and with Django's admin style.

Here are some recommendations for making your app work as seamlessly as possible:

- Try avoiding using `color`, `background-color`, or other color styles where possible and meaningful.
- If necessary, use as few as possible standard django CMS colors (preferably from the list below with plain Django fallback colors)
- Use the following pattern: `var(--dca-color-var, var(--fallback-color-var, #xxxxxx))` where `#xxxxxx` represents the light version of the color. This tries django CMS color scheme first and falls back to Django color scheme if djangocms-simple-admin-style is not available.
- Avoid media queries like `@media (prefers-color-scheme: dark)` since they would ignore forced settings to light or dark.

The admin frontend pulls the style from django admin styles and - if present - from djangocms-simple-admin-style. Django itself also uses CSS variables to implement admin mode, these can be used as dark mode-aware fall-back colors.

Here's a table of django CMS' CSS color variables and their Django fallbacks:

Variable name	Color	Fallback	Color
--dca-white	#ffffff	--body-bg	#ffffff
--dca-gray	#666	--body-quiet-color	#666
--dca-gray-lightest	#f2f2f2	--darkened-bg	#f8f8f8
--dca-gray-lighter	#ddd	--border-color	#ccc
--dca-gray-light	#999	--close-button-bg	#888
--dca-gray-darker	#454545		
--dca-gray-darkest	#333		
--dca-gray-super-lightest	#f7f7f7		
--dca-primary	#00bbff	--primary	#79aec8
--dca-black	#000	--body-fg	#303030

This leaves these recommendations for color scheme dependent colors:

```
white:          var(--dca-white, var(--body-bg, #fff))
gray:           var(--dca-gray, var(--body-quiet-color, #666))
gray-lightest:  var(--dca-gray-lightest, var(--darkened-bg, #f2f2f2))
gray-lighter    var(--dca-gray-lighter, var(--border-color, #ddd))
gray-light:     var(--dca-gray-lightest, var(--darkened-bg, #f2f2f2))
gray-primary:   var(--dca-primary, var(--primary, #0bf))
black:          var(--dca-black, var(--body-fg), #000))
```

5.3.3 How-to guides

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

Installation & deployment

Install django CMS with Docker

The django CMS quickstart project is a Docker-based production-ready setup. If you know your way around Docker, you will be able to quickly set up a project that is ready for deployment.

For a simpler local setup without Docker, see *Installing django CMS*.

Prerequisites

Install Docker from docker.com. If you have not used Docker before, read the Docker getting started guide.

Setup

Open a terminal and navigate to your projects folder:

```
git clone git@github.com:django-cms/django-cms-quickstart.git
cd django-cms-quickstart
docker compose build web
docker compose up -d database_default
docker compose run web python manage.py migrate
docker compose run web python manage.py createsuperuser
docker compose up -d
```

Visit http://localhost:8000/admin and log in with your superuser credentials.

The quickstart project

The quickstart project is a minimal Django project with production-ready defaults including:

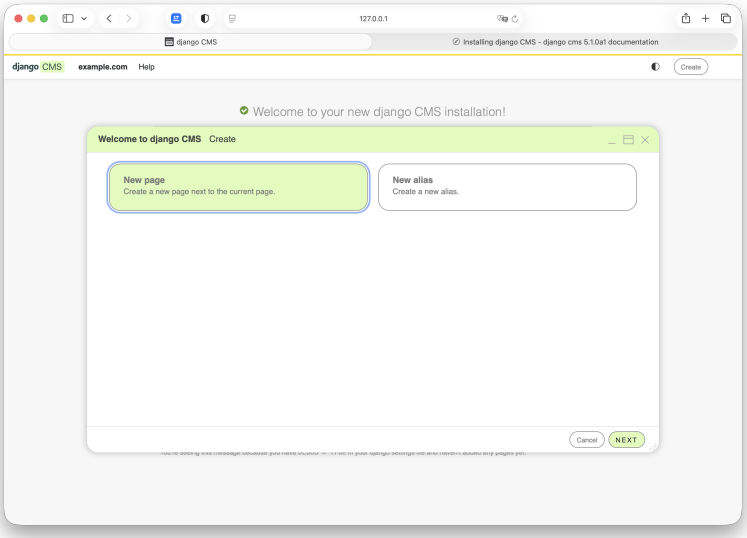
- PostgreSQL database configuration
- Static and media file handling
- Production-ready settings structure
- Docker Compose configuration for development and production

For more details, see the quickstart repository README.

Creating pages

Once logged in:

1. Press **Create** on the top right
2. Select **New Page** and press **Next**
3. Add a title and content, then click **Create**
4. Press **Publish** to make the page visible



Next steps

- Continue with the *Tutorial* to learn django CMS development
- See *Install django CMS manually* to understand the configuration in detail

Install django CMS manually

This how-to guide walks you through manually installing and configuring django CMS, showing every setting that the automated `djangocms` command otherwise writes for you. If you prefer the automated setup, follow the *installation tutorial* for a new project, or *Add django CMS to an existing project* for an existing one.

This guide assumes you have basic familiarity with Python and Django.

Set up a project

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

Important

We strongly recommend doing all of the following steps in a virtual environment.

```
python3 -m venv .venv  # create a virtualenv
source .venv/bin/activate  # activate it
pip install --upgrade pip  # Upgrade pip
```

Then install django CMS:

```
pip install django-cms
```

If you are starting from scratch, create a plain Django project first:

```
django-admin startproject myproject
cd myproject
```

If you are adding django CMS to an existing project, work from your project's root directory (the one containing `manage.py`) instead. All of the following steps apply to both cases.

Required configuration

To add django CMS to a Django project, you need to install dependencies and modify `settings.py` and `urls.py`.

Tip

The steps in this section can be automated by running `djangocms .` from your project directory — see *Add django CMS to an existing project*. The rest of this guide describes how to perform the same steps by hand.

Install required packages

Add django CMS and its dependencies to your requirements file:

```
django-cms>=4.1
django-sekizai
django-treebeard
```

Or install directly:

```
pip install django-cms
```

For a fully-featured setup, also install recommended plugins:

```
pip install djangocms-text djangocms-frontend django-filer djangocms-versioning␣
↪djangocms-alias
```

INSTALLED_APPS

Add to `INSTALLED_APPS` (order matters):

```
INSTALLED_APPS = [
    # Add before django.contrib.admin for admin styling (optional)
    "djangocms_simple_admin_style",

    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "django.contrib.sites",  # Required by django CMS

    # django CMS core
    "cms",
    "menus",
    "treebeard",
    "sekizai",

    # Recommended plugins (if installed)
    "filer",
    "easy_thumbnails",
    "djangocms_text",
    "djangocms_frontend",
    "djangocms_frontend.contrib.grid",
    "djangocms_frontend.contrib.image",
    "djangocms_frontend.contrib.link",
    "djangocms_versioning",
    "djangocms_alias",

    # Your existing apps
    # ...
]
```

- django CMS needs Django's `django.contrib.sites` framework
- `cms` and `menus` are the core django CMS modules
- django-treebeard manages the page tree
- django-sekizai handles CSS/JS blocks in templates

Required settings

Add to `settings.py`:

```
SITE_ID = 1

X_FRAME_OPTIONS = "SAMEORIGIN"
```

Language settings

django CMS requires the `LANGUAGES` setting:

```
LANGUAGES = [
    ("en", "English"),
    ("de", "German"),
    ("it", "Italian"),
]
LANGUAGE_CODE = "en"
```

Database

django CMS requires a relational database. SQLite works for development:

```
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.sqlite3",
        "NAME": BASE_DIR / "db.sqlite3",
    }
}
```

For production, use PostgreSQL, MySQL, or MariaDB:

```
pip install psycopg2     # for PostgreSQL
pip install mysqlclient  # for MySQL or MariaDB
```

MIDDLEWARE

Add to `MIDDLEWARE` (order matters):

```
MIDDLEWARE = [
    "cms.middleware.utils.ApphookReloadMiddleware",  # Optional, must be first
    "django.middleware.security.SecurityMiddleware",
    # ... existing middleware ...
    "django.middleware.locale.LocaleMiddleware",  # After SessionMiddleware
    # ... existing middleware ...
    "cms.middleware.user.CurrentUserMiddleware",
    "cms.middleware.page.CurrentPageMiddleware",
    "cms.middleware.toolbar.ToolbarMiddleware",
    "cms.middleware.language.LanguageCookieMiddleware",
]
```

`ApphookReloadMiddleware` is optional but recommended for *apphook reloading*.

TEMPLATES

Add `sekizai` to `INSTALLED_APPS` and configure context processors:

```
TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",
        "DIRS": [BASE_DIR / "templates"],
```

(continues on next page)

(continued from previous page)

```
        "APP_DIRS": True,
        "OPTIONS": {
            "context_processors": [
                # ... existing context processors ...
                "django.template.context_processors.i18n",
                "sekizai.context_processors.sekizai",
                "cms.context_processors.cms_settings",
            ],
        },
    },
]
```

CMS_TEMPLATES

django CMS requires at least one template. Add *CMS_TEMPLATES* to settings:

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

Create a `templates` directory and add `home.html`:

```
{% load cms_tags sekizai_tags %}
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>{% page_attribute "page_title" %}</title>
    {% render_block "css" %}
</head>
<body>
    {% cms_toolbar %}
    <main>
        {% placeholder "content" %}
    </main>
    {% render_block "js" %}
</body>
</html>
```

Key template tags:

- `{% load cms_tags sekizai_tags %}` loads required template tag libraries
- `{% page_attribute "page_title" %}` extracts the page's title
- `{% render_block "css" %}` and `{% render_block "js" %}` load CSS and JavaScript
- `{% cms_toolbar %}` renders the django CMS toolbar
- `{% placeholder "content" %}` defines where plugins can be inserted

If using django-filer, add thumbnail configuration:

```
THUMBNAIL_HIGH_RESOLUTION = True
THUMBNAIL_PROCESSORS = (
```

(continues on next page)

(continued from previous page)

```
    "easy_thumbnails.processors.colorspace",
    "easy_thumbnails.processors.autocrop",
    "filer.thumbnail_processors.scale_and_crop_with_subject_location",
    "easy_thumbnails.processors.filters",
)
```

urls.py

Add CMS URLs using `i18n_patterns`:

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

urlpatterns = [
    # Non-i18n URLs (if any)
]

urlpatterns += i18n_patterns(
    path("admin/", admin.site.urls),
    # Your existing app URLs
    # path("myapp/", include("myapp.urls")),
    # CMS URLs - must be last (catch-all)
    path("", include("cms.urls")),
)

# For development only
if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Media files

Configure media file handling in `settings.py`:

```
MEDIA_URL = "/media/"
MEDIA_ROOT = BASE_DIR / "media"
```

For production, configure proper media file serving per Django's documentation.

Run migrations

Create database tables and verify configuration:

```
python manage.py migrate
python manage.py createsuperuser  # If you don't have one
python manage.py cms check
python manage.py runserver
```

Visit http://127.0.0.1:8000 and log in to start creating pages.

Optional plugins

django CMS uses plugins to handle content. The `djangocms` command installs recommended plugins automatically. If you installed manually, consider adding these.

Django Filer

Django Filer provides file and image management.

```
pip install django-filer>=3.0
```

Add to `INSTALLED_APPS`:

```
"filer",
"easy_thumbnails",
```

djangocms-text

djangocms-text provides rich text editing.

```
pip install djangocms-text
```

Add `djangocms_text` to `INSTALLED_APPS`.

djangocms-frontend

djangocms-frontend adds Bootstrap 5 support.

```
pip install djangocms-frontend
```

Add to `INSTALLED_APPS`:

```
"djangocms_frontend",
"djangocms_frontend.contrib.accordion",
"djangocms_frontend.contrib.alert",
"djangocms_frontend.contrib.badge",
"djangocms_frontend.contrib.card",
"djangocms_frontend.contrib.carousel",
"djangocms_frontend.contrib.collapse",
"djangocms_frontend.contrib.content",
"djangocms_frontend.contrib.grid",
"djangocms_frontend.contrib.image",
"djangocms_frontend.contrib.jumbotron",
"djangocms_frontend.contrib.link",
"djangocms_frontend.contrib.listgroup",
"djangocms_frontend.contrib.media",
"djangocms_frontend.contrib.tabs",
"djangocms_frontend.contrib.utilities",
```

djangocms-versioning and djangocms-alias

Install for publishing workflow and reusable content blocks:

```
pip install djangocms-versioning djangocms-alias
```

Add to `INSTALLED_APPS`:

```
"djangocms_versioning",
"djangocms_alias",
```

Run migrations

After installing plugins:

```
python manage.py migrate
```

Verify your installation

Use django CMS's built-in check command to verify your configuration:

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

This checks your configuration, applications, and database, reporting any problems. Run it after each configuration step to verify progress.

Next steps

- Read the user guide for a walk-through of basics
- Follow the *tutorials for developers*
- See Django deployment documentation for production

Add django CMS to an existing project

You can add django CMS to an existing Django project with a single command:

```
pip install django-cms
djangocms .
```

Run it from the project root (the directory containing `manage.py`), with your project's virtual environment activated.

Warning

The command makes automated, best-effort edits to your settings and urls files. Make sure your project is under version control (or backed up) so you can review every change afterwards. The command asks for confirmation before changing anything.

Tip

Run it with `--dry-run` first to preview every change as a unified diff, without writing any files or installing packages:

```
djangocms . --dry-run
```

What the command does

The command reads the settings module from `manage.py` and updates your project files:

- adds the django CMS apps (and those of the selected add-ons) to `INSTALLED_APPS`,
- adds the django CMS middleware and template context processors,
- appends required settings such as `SITE_ID`, `LANGUAGES` (derived from your `LANGUAGE_CODE`) and `CMS_TEMPLATES` if they are missing,
- adds the django CMS url patterns to your `ROOT_URLCONF`,
- creates a `templates` directory with a minimal base template if your project has none yet.

Existing entries are never duplicated, so the command is safe to run again.

It then lists the required packages and asks whether to install them. If you agree, it installs them, runs the database migrations and validates the installation with `cms check`.

Choosing options

The same options as for creating a new project are available, for example:

```
djangocms . --mode headless --no-versioning
```

selects the headless mode (content is served through a REST API instead of HTML pages) and skips content versioning. See the *djangocms command reference* for all options and their defaults.

Review the changes

The edits are best-effort: projects with heavily customized settings (for example, split settings modules or settings computed at runtime) may need manual adjustments. After running the command:

- review the diff of your settings and urls files,
- start the development server with `python -m manage runserver`,
- log in with an existing superuser account and check that the toolbar appears.

No superuser is created when installing into an existing project; create one with `python -m manage createsuperuser` if needed.

Prefer full control?

If you would rather apply every change yourself, follow *the manual configuration steps*, which walk through each setting the command writes for you.

Using core functionality

How to use placeholders outside the CMS

Placeholder fields 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.

Changed in version 4.0: Since django CMS 4.0 the toolbar offers preview and edit endpoints for Django models which contain Placeholders.

- This allows for models (such as django CMS Alias) which do not have a user-facing view to still contain placeholders.

- However, it requires the registration of frontend-editable models with django CMS.
- Also, views need to tell the toolbar if they contain a frontend-editable model.

Placeholders can be viewed as containers for *CMSPlugin* instances, and can be used outside the CMS in custom applications using the *PlaceholderRelationField*.

By defining a *PlaceholderRelationField* on a custom model you can take advantage of the full power of *CMSPlugin* in one or more placeholders.

> **Warning**
>
> Django CMS 3.x used a different way of integrating placeholders. It's `PlaceholderField("slot_name")` needs to be changed into a `PlaceholderRelationField` (available since django CMS 4.x).

Two ways to render model placeholders

A quick glossary:

- A **slot** is the string name that identifies a placeholder in a template (for example `"content"`). Slot names are also used by *CMS_PLACEHOLDER_CONF* to configure which plugins can be inserted.
- A **placeholder** is the per-instance container (a *Placeholder* object) that holds the plugins for a given slot on a given model instance.
- The **structure board** is django CMS's frontend editor view where editors add, remove and reorder plugins.

django CMS offers two template tags for placeholders on your own models:

- *placeholder* — *declares and renders* in one step. The same template is both what your view renders and what django CMS scans to discover slot names for the structure board.
- *render_placeholder* — *only renders* a placeholder instance that the model exposes as a property (typically via *get_placeholder_from_slot()*). Because this tag does not declare its slot, the model also needs a separate declaration-only template for the structure board to discover the slots.

Prefer Approach 1 for new models. Use Approach 2 only when you're integrating with code that already uses *render_placeholder*, or when you want each placeholder exposed as a named model property.

Both approaches share the same *PlaceholderRelationField* and the toolbar integration described in *Setting and getting the placeholder-enabled object from the toolbar* below.

Approach 1: Using `{% placeholder %}`

Step 1 — Add a *PlaceholderRelationField* and a `get_template()` method pointing to the template you'll create in Step 2. This is the template your view will render and that django CMS will scan for slot declarations:

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

class MyModel(models.Model):
    # your fields
    placeholders = PlaceholderRelationField()

    def get_template(self):
        return "my_app/my_model_template.html"
```

Step 2 — Create that template. It declares the slots your model owns with the *placeholder* tag, alongside any other markup you need — the same tags are used both to declare the slots (so django CMS can list them in the structure board) and to render their content:

```
{# templates/my_app/my_model_template.html #}
{% load cms_tags %}
<h1>{{ object.title }}</h1>
<main>
    {% placeholder "content" %}
</main>
<aside>
    {% placeholder "sidebar" %}
</aside>
```

Step 3 — In your view, register the model instance on the toolbar (see *Setting and getting the placeholder-enabled object from the toolbar*) and render the same template. Setting the toolbar object is what tells the *placeholder* tag which object's placeholders to resolve against:

```
from django.shortcuts import get_object_or_404, render

def my_model_detail(request, id):
    obj = get_object_or_404(MyModel, id=id)
    request.toolbar.set_object(obj)
    return render(request, "my_app/my_model_template.html", {"object": obj})
```

Load the view with the frontend editor active to add plugins to each slot — see *Adding content to a placeholder* below.

Approach 2: Using `{% render_placeholder %}`

In this approach the model exposes a *Placeholder* instance per slot, and the user-facing template passes that instance to *render_placeholder*. Because *render_placeholder* does not declare its slot, the model also needs a **separate declaration-only template**. Unlike Approach 1, the template registered with `get_template()` is *not* the one your view renders.

Step 1 — Add the *PlaceholderRelationField*, a cached-property accessor for each slot, and a `get_template()` pointing to the declaration-only template you'll create in Step 2:

```
from django.db import models
from django.utils.functional import cached_property
from cms.models.fields import PlaceholderRelationField
from cms.utils.placeholder import get_placeholder_from_slot

class MyModel(models.Model):
    # your fields
    placeholders = PlaceholderRelationField()

    def get_template(self):
        return "my_app/my_model_structure.html"

    @cached_property
    def content_placeholder(self):
        return get_placeholder_from_slot(self.placeholders, "content")

    @cached_property
```

(continues on next page)

(continued from previous page)

```
    def sidebar_placeholder(self):
        return get_placeholder_from_slot(self.placeholders, "sidebar")
```

get_placeholder_from_slot() retrieves — or, if needed, creates — the placeholder object for the given slot name. Add one cached property per slot.

Step 2 — Create the declaration-only template. This file is never rendered to the visitor; django CMS only scans it for *placeholder* tags to discover which slots the model owns. Other markup in it is ignored:

```
{# templates/my_app/my_model_structure.html #}
{% load cms_tags %}
{% placeholder "content" %}
{% placeholder "sidebar" %}
```

Step 3 — Create the user-facing template. Pass each placeholder instance from Step 1 to *render_placeholder*:

```
{# templates/my_app/my_model_template.html #}
{% load cms_tags %}
<h1>{{ object.title }}</h1>
<main>
    {% render_placeholder object.content_placeholder %}
</main>
<aside>
    {% render_placeholder object.sidebar_placeholder %}
</aside>
```

See *render_placeholder* for optional parameters (`width`, `language`, `as`).

Step 4 — In your view, register the model instance on the toolbar (see *Setting and getting the placeholder-enabled object from the toolbar*) and render the user-facing template:

```
from django.shortcuts import get_object_or_404, render

def my_model_detail(request, id):
    obj = get_object_or_404(MyModel, id=id)
    request.toolbar.set_object(obj)
    return render(request, "my_app/my_model_template.html", {"object": obj})
```

Load the view with the frontend editor active to add plugins to each slot — see *Adding content to a placeholder* below.

Setting and getting the placeholder-enabled object from the toolbar

The toolbar provides two methods for managing the object associated with placeholder editing. These are essential for enabling the toolbar's Edit and Preview buttons, and — in Approach 1 — for resolving *placeholder* tags against your model.

set_object(obj)
: Associates a Django model instance with the toolbar. This method only sets the object if one hasn't already been set. The object is typically a model instance that contains placeholders, such as a *PageContent* object or any other model that supports editable placeholders through a *PlaceholderRelationField*.

  The associated object is used by other toolbar methods to generate appropriate URLs for editing, preview, and structure modes.

get_object()
: Returns the object currently associated with the toolbar, or `None` if no object has been set. This method can be

used to retrieve the object that was previously set using `set_object()`.

Usage in Views

If the object has a user-facing view it typically is identical to the preview and editing endpoints, but has to get the object from the URL (e.g., by its primary key). **It also needs to set the toolbar object, so that the toolbar will have Edit and Preview buttons:**

```
from django.shortcuts import get_object_or_404, render


def render_my_model(request, obj):
    return render(
        request,
        "my_model_detail.html",
        {
            "object": obj,
        },
    )


def my_model_detail(request, id):
    obj = get_object_or_404(MyModel, id=id)  # Get the object (here by id)
    request.toolbar.set_object(obj)  # Announce the object to the toolbar
    return render_my_model(request, obj)  # Same as preview rendering
```

You can also retrieve the object from the toolbar in your views using the `get_object()` method:

```
def my_view(request):
    my_content = request.toolbar.get_object()  # Can be anything: PageContent,␣
↪PostContent, AliasContent, etc.
    if my_content:
        my_post = my_content.post  # only works for PostContent, of course
    # ... rest of your view logic
```

Note

If using class based views, you can set the toolbar object in the `get_context_data` method of your view and add a stub view usable when you *register the model for frontend editing*.

```
from django.views.generic.detail import DetailView

class MyModelDetailView(DetailView):
    # your detail view attributes

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        self.request.toolbar.set_object(self.object)
        return context

def my_model_endpoint_view(request, my_model):
    return MyModelDetailView.as_view()(request, pk=my_model.pk)
```

Usage in Templates

You can also access the toolbar object directly in templates:

```
{# Access the object directly #}
{{ request.toolbar.get_object.title }}

{# Use with template tag for more complex operations #}
{% with my_obj=request.toolbar.get_object %}
    {% if my_obj %}
        <h2>{{ my_obj.title }}</h2>
        <p><strong>{{ my_obj.description }}</strong></p>
    {% endif %}
{% endwith %}
```

Note

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 the slots to the model

To let django CMS' frontend editor know which placeholders the model contains, declare them in a second template, only needed for rendering the structure mode, called, say, `templtes/my_app/my_model_structure.html`:

```
{% load cms_tags %}
{% placeholder "slot_name" %}
```

The important bit is to include all slot names for the model in the structure template. Other parts of the templte are not necessary.

Add the structure mode template to the model

Let the model know about this template by declaring the `get_template()` method:

```
class MyModel(models.Model):
    ...

    def get_template(self):
        return "my_app/my_model_structure.html"

    ...
```

Registering the model for frontend editing

Added in version 4.0.

The final step is to register the model for frontend editing. Since django CMS 4 this is done by adding a *CMSAppConfig* class to the app's *cms_config.py* file:

```
from cms.app_base import CMSAppConfig
from . import models, views


class MyAppConfig(CMSAppConfig):
    cms_enabled = True
    cms_toolbar_enabled_models = [(models.MyModel, views.render_my_model)]
```

Note

If using class based views, use the stub view in `cms_toolbar_enabled_models` attribute.

```
cms_toolbar_enabled_models = [(models.MyModel, views.my_model_endpoint_view)]
```

Adding content to a placeholder

Placeholders can be edited from the frontend by visiting the page displaying your model (where you put the *placeholder* or *render_placeholder* tag), then appending `?toolbar_on` 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.

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 *PlaceholderRelationField*, 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 `PlaceholderRelationField` 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:
            return True
    return False
```

How to serve multiple languages

If you used django CMS quickstart to start your project, you'll find that it's already set up for serving multilingual content. Our installation guide also does the same.

This guide specifically describes the steps required to enable multilingual support, in case you need to do 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.urls.i18n import i18n_patterns
from django.contrib import admin
from django.contrib.staticfiles.urls import staticfiles_urlpatterns
from django.urls import include, path
from django.views.i18n import JavaScriptCatalog


admin.autodiscover()

urlpatterns = i18n_patterns(
    re_path('jsi18n/', JavaScriptCatalog.as_view(), name='javascript-catalog'),
)
urlpatterns += staticfiles_urlpatterns()

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

Monolingual URLs

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

```
urlpatterns += [
    path('admin', admin.site.urls),
    path('', 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` 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().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" %}
```

A common real-world example is adding these links in the <head> of your page template so search engines can read them as page metadata. This helps crawlers understand that the URLs are language alternatives of the same content, which improves language targeting in search results and reduces duplicate-content ambiguity for multilingual pages.

```
{% load cms_tags %}
{% for language in request.current_page.get_languages %}
    <link rel="alternate" hreflang="{{ language }}" href="{% page_language_url language
↪%}">
{% endfor %}
```

Configuring language-handling behaviour

CMS_LANGUAGES is the setting that controls how django CMS serves content across multiple languages — per-language fallbacks, `redirect_on_fallback`, `hide_untranslated`, the `public` flag, and per-site language configuration in

multi-site projects. For the authoritative options, see *Configuring django CMS*; for the *why* behind each option (fallback semantics, URL strategies, per-language menus), see *multilingual support* in the Explanation section.

Custom Language Resolution

You can override django-cms's language resolution by setting `request.LANGUAGE_CODE` in your own middleware. Common use cases include:

- Serving different languages based on domains (e.g., `example.com` → English, `example.de` → German):

```
class DomainLanguageMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        host = request.get_host().split(':')[0]
        if host.endswith('.de'):
            request.LANGUAGE_CODE = 'de'
        elif host.endswith('.fr'):
            request.LANGUAGE_CODE = 'fr'
        return self.get_response(request)
```

- Setting language based on user preferences or geolocation
- Implementing custom language negotiation logic

You can customize the default language for each site using the *CMS_LANGUAGES* setting.

Multi-Site Installation

django CMS uses Django's sites framework to manage multi-site installations.

There are three ways to run multiple sites with django CMS:

1. The classic (and most common) approach: separate settings files with distinct `SITE_ID` values.
2. Since django CMS 5.1: shared settings without a `SITE_ID` setting; the active site is resolved by Django's Sites framework based on the request's hostname.
3. Since django CMS 5.1: a custom site middleware; if it sets `request.site`, django CMS will honor it.

Approach 1: Separate settings with SITE_ID (classic)

Operate multiple websites in the same virtualenv by using copies of `manage.py` and `wsgi.py`, plus per-site settings and URL configuration. You can use the same database for all sites or separate databases for stricter isolation.

A common pattern is to keep shared configuration in a base settings file (e.g. `my_project/base_settings.py`), which is imported from each site-specific settings file and overridden as needed. Optionally, import local, unversioned overrides at the end (e.g. secrets, database credentials).

1. Copy and edit `wsgi.py` and `manage.py` (e.g. to `wsgi_second_site.py` and `manage_second_site.py`) and point `DJANGO_SETTINGS_MODULE` to the site's settings module:

```
os.environ.setdefault(
    "DJANGO_SETTINGS_MODULE",
    "my_project.settings_second_site"
)
```

2. In each site-specific settings module, import the shared base and override site-specific values:

```
# my_project/settings_second_site.py
from .base_settings import *

SITE_ID: int = 2
ROOT_URLCONF: str = 'my_project.urls_second_site'
# other site-specific settings...

from .settings_local import *  # optional, not under version control
```

3. Configure your web server so the site uses its dedicated `wsgi_*.py` (e.g. `wsgi_second_site.py`).

Note

You do not have to include Django's admin on all sites. You can run a site without admin by not exposing the admin URLs in that site's URL configuration. This is useful if you only want to expose the frontend editing and preview endpoints on certain sites, e.g., for security reasons.

The toolbar will only be available on sites which do have admin endpoints.

Approach 2 (since 5.1): Shared settings without SITE_ID

If you omit `SITE_ID` from settings, django CMS determines the active site at request time using the Sites framework and the incoming request's hostname. This avoids duplicating `manage.py`, `wsgi.py`, and settings modules.

Requirements and notes:

- Create one `Site` object per domain in Django's admin with the correct `domain`.
- Ensure all domains are listed in `ALLOWED_HOSTS`.
- `CMS_LANGUAGES` can still define per-site language sets (by numeric site ID) and a `'default'` block. The active site's language configuration is selected at runtime.

Example (single shared `settings.py`):

```
# Do not set SITE_ID here

ALLOWED_HOSTS = ['site1.example.com', 'site2.example.com']

CMS_LANGUAGES = {
    1: [
        {'code': 'en', 'name': 'English', 'public': True},
    ],
    2: [
        {'code': 'de', 'name': 'Deutsch', 'public': True},
    ],
    'default': {
        'public': True,
        'hide_untranslated': False,
    },
}
```

> **Note**
>
> Management commands that run without a request (and thus without a hostname) may still need an explicit site context. Provide `--site` options where available, set `SITE_ID` just for the command's runtime, or set `request.site` in custom command logic.

Approach 3 (since 5.1): Custom site middleware

You can provide middleware that sets `request.site`. django CMS will respect this attribute as the current site for the request, regardless of whether `SITE_ID` is configured.

Example middleware:

```
# my_project/middleware.py
from django.contrib.sites.models import Site

class CurrentSiteFromHostMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        host = request.get_host().split(':')[0]
        try:
            request.site = Site.objects.get(domain=host)
        except Site.DoesNotExist:
            request.site = None  # optional fallback
        return self.get_response(request)
```

Register the middleware (early in the stack is recommended):

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'my_project.middleware.CurrentSiteFromHostMiddleware',
    # ...
]
```

Apphooks and per-request site resolution

Apphooks are directly linked into Django's URL patterns. Since a common settings file results in common URL patterns, django CMS adds apphooks of all sites into the same URL pattern **if the settings contain no** `SITE_ID`.

To isolate apphooks, the `cms_site_filter()` decorator is automatically added to the apphook view functions, which ensures that an apphook is only displayed on its designated site.

However, when multiple sites are configured, this can lead to apphooks shadowing each other if they share the same URL path. It is important to manage URL patterns carefully to avoid conflicts between apphooks across different sites.

Language and URL configuration tips

- Ensure each domain has a matching `Site` with the exact `domain` value.
- Configure language sets in `CMS_LANGUAGES` per site or via the `'default'` block as needed.
- With approaches 2 and 3, the active site is determined per request; for batch tasks without requests, pass an explicit site context or temporarily set `SITE_ID`.

How to work with templates

django CMS uses Django's template system to manage the layout of the CMS pages.

Django's Template System

Django's template language is designed to strike a balance between power and ease. It's designed to feel comfortable to those used to working with HTML. If you have any exposure to other text-based template languages, such as Smarty or Jinja2, you should feel right at home with Django's templates.

The template system, out of the box, should be familiar to those who have worked with desktop publishing or web design. Tags are surrounded by {% and %} and denote the actions like loops and conditionals. Variables are surrounded by {{ and }} and get replaced with values when the template is rendered.

Learn more about Django's template system in the Django documentation.

Django CMS and Django's Template System

Django templates

You are totally free on how to name your templates, but we encourage you to use the general Django conventions, including letting all templates inherit from a base template by using the `extends` template tag or putting templates in a folder named after the application providing it.

> **Note**
>
> Some django CMS apps, like django CMS Alias, assume the base template is called `base.html`. If you happen to prefer a different name for the base template and need to use such apps, you can create a `base.html` template that just consists of the `{% extends "your_base_template.html" %}` tag.

A fresh installation of django CMS using the quickstarter project or the `djangocms` command comes with a default template that for reasons of convenience is provided by django CMS frontend and based on Bootstrap. We encourage you to create your own templates as you would do for any Django project.

Generally speaking, django CMS is wholly frontend-agnostic. It doesn't care what your site's frontend is built on or uses: You are free to decide which CSS framework or JS library to use (if any).

When editing, the frontend editor will replace part of the current document's DOM. This might require some JS widgets to be reinitialized. See *Frontend integration* for more information.

CMS templates

You need to configure which templates django CMS should use. You can do this by either setting the *CMS_TEMPLATES* or the *CMS_TEMPLATES_DIR* settings.

You can select the template by page (and language) in the page menu of django CMS' toolbar. By default, a new page uses the same template as its parent page (this is template *assignment*, not Django `{% extends %}` inheritance — see *Template inheritance pattern* below). A root page uses the first template in *CMS_TEMPLATES* if no other template is explicitly set.

To work seamlessly with django CMS, **your templates should include the `{% cms_toolbar %}` tag right as the first item in your template's** <body>. This tag will render the toolbar for logged-in users.

> **Note**
>
> The toolbar can also be displayed in views independent of django CMS. To provide a consistent user experience, many projects include the toolbar in their base template and share it with the whole Django project.

Also, you need to tell django CMS where to place the content of your pages. This is done using **placeholders**. A placeholder is a named area in your template where you can add content plugins. You can add as many placeholders as you want to your templates.

To add a placeholder to your template, use the `{% placeholder "name" %}` template tag. The name is the name of the template slot. It will be shown in the structure board of the frontend editor. Typical names are "main", "sidebar", "footer", etc.

Finally, you need to add `{% render_block "css" %}` in the `<head>` section of your CMS templates and `{% render_block "js" %}` right before the closing `</body>` tag of your CMS templates. This will render the CSS and JavaScript at the appropriate places in your CMS templates.

django CMS uses django-sekizai to manage CSS and JavaScript. To use the sekizai tags, you need to load the `sekizai_tags` template tags in your template: `{% load sekizai_tags %}`.

Template inheritance pattern

django CMS projects typically organize their templates into three layers, using Django's standard `{% extends %}` and `{% block %}` mechanics:

1. **Base template** (usually `base.html`) — the shared chrome: `<!DOCTYPE>`, `<head>`, `<body>`, `{% cms_toolbar %}`, and the sekizai `{% render_block "css" %}` / `{% render_block "js" %}` tags. It defines empty `{% block %}` slots that per-page content will fill.
2. **CMS page template** — extends `base.html` and fills those blocks with *`placeholder`* tags for page-specific content. These are the templates you list in *`CMS_TEMPLATES`* and select in the page's *Advanced settings*.
3. **Apphook or model template** — extends `CMS_TEMPLATE` (see *CMS_TEMPLATE* below) and overrides blocks with application output. This way an app view inherits whichever page template the current CMS page is using — including the toolbar, sekizai blocks, and any surrounding layout — without hardcoding a base template path.

Layers 1 and 2 are plain Django inheritance. Layer 3 is the django CMS-specific trick: `CMS_TEMPLATE` is a context variable holding the current CMS page's template path, so `{% extends CMS_TEMPLATE %}` reuses whichever template the CMS is currently rendering.

Where to put placeholders

- **Page-specific** content (main content area, article body) → the CMS page template (layer 2). Each CMS page template declares its own slot set.
- **Site-wide** content that is identical on every page (a promotional banner shared across all pages) → declare once in the base template (layer 1) so it renders everywhere, or use a `static_alias` (see *Static aliases* below) if editors should manage it centrally from Django admin.
- **Apphook or model-specific** content → the app's template (layer 3); the placeholder resolves against the current page or the toolbar object.

> **Warning**
>
> If you put a *`placeholder`* inside a `{% block %}` that a child template overrides without calling `{% block.super %}`, the placeholder disappears from the rendered page. Keep shared placeholders *outside* overridable blocks, or

make sure children call `{% block.super %}` when they mean to keep the base content.

Example

Here is an example of a simple template that uses placeholders:

```
{% extends "base.html" %}
{% load cms_tags djangocms_alias_tags %}
{% block title %}{% page_attribute "page_title" %}{% endblock title %}
{% block content %}
    <header>
        {% placeholder "header" %}
    </header>
    <main>
        {% placeholder "main" %}
    </main>
    <footer>
        {% static_alias "footer" %}
    </footer>
{% endblock content %}
```

In this example the child template:

- extends `base.html` — reusing the shared chrome without repeating it;
- fills `{% block title %}` and `{% block content %}` — these bodies replace the empty blocks of the same name in the base template;
- declares three placeholders (`header`, `main`, `footer`) inside `{% block content %}` — each becomes an editable slot in the structure board.

Everything in the base template *outside* those blocks — `<html>`, `<head>`, `{% cms_toolbar %}`, the sekizai `{% render_block %}` tags — is inherited unchanged.

The underlying base template could look like this:

```
{% load cms_tags sekizai_tags %}
<!DOCTYPE html>
<html>
    <head>
        <title>{% block title %}{% endblock title %}</title>
        {% render_block "css" %}
    </head>
    <body>
        {% cms_toolbar %}
        {% block content %}{% endblock content %}
        {% render_block "js" %}
    </body>
</html>
```

Static aliases

Added in version 4.0.

> **Note**
>
> Using `static_alias` requires the installation of djangocms-alias to work.

The package djangocms-alias provides an admin page in Django admin where special types of placeholders called "static aliases" can be managed and its contents edited.

Frequent Use Cases:

1. Editors wish to manage repeated content centrally (DRY - don't repeat yourself)
2. Developers wish to add CMS functionality to their custom application's templates

Repeated content: Often, content areas such as a footer, a header or a sidebar have identical content across all pages of a website. djangocms-alias provides a Django admin page for editors to manage such general site-wide content in one place.

Custom applications: Templates in custom applications usually follow some well-defined business logic which is normally hard-coded in the template. However the same templates might include areas of "static" content, i.e. content that editors wish to manage. The django CMS *`placeholder`* tag works in templates attached to the django CMS *`Page`* model, or in templates rendered by apphook views (see *Using placeholders on an apphooked page*). For content areas that are not tied to a page or apphook, djangocms-alias closes the gap by providing editors central access to such custom content areas.

CMS_TEMPLATE

`CMS_TEMPLATE` is a context variable whose value is the template path for the current CMS page (or apphook page), or the first template in *`CMS_TEMPLATES`* when the URL is not managed by the CMS.

Use it as the base for any app template that should blend into CMS pages — write `{% extends CMS_TEMPLATE|default:"my_default_template.html" %}` instead of hardcoding a base-template path. The app template will then pick up the toolbar, sekizai blocks, and any layout the current page template provides, even if editors later switch the page to a different template.

Without `CMS_TEMPLATE`, an app template with a hardcoded `{% extends "myproject/base.html" %}` would always render with `myproject/base.html` regardless of which page template the CMS is rendering — causing layout drift whenever editors change the page's template.

Example: cms template

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

Example: application template

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

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

Template assignment across translations and versions

This section is about which template file is *assigned* to a page content, not about Django's `{% extends %}` / `{% block %}` inheritance (see *Template inheritance pattern* for that).

Default behavior (without versioning)

In django CMS (version 4 and above), when creating page content translations, all content is immediately published. When creating a translation of an existing page, the template from the original page content is copied to the new translation. This behavior can be seen in the *create_translation* method. The template is explicitly copied from the original page content to ensure consistent layout across languages.

Note for django CMS versions prior to 4.2:

In older versions of django CMS (prior to 4.2), when creating page content translations, the template is not automatically copied from the original page content. Instead, the default site template (or the template from the last published version) is applied, and manual adjustments may be necessary to ensure consistency.

Behavior with djangocms-versioning

When using djangocms-versioning, pages and their content can exist in different states (draft, published, etc.). This introduces some nuances in template inheritance:

1. When you create a page and publish it, the template is set and published
2. When creating a translation of a page content, the new translation will inherit the template from the draft version of the main language page content if one exists
3. If no draft version exists in the main language, the template will be inherited from the published version instead

This behavior ensures that new translations automatically pick up any template changes that are in progress (draft state) in the main language, maintaining consistency across translations even before changes are published.

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=[
        'django.middleware.cache.UpdateCacheMiddleware',
```

(continues on next page)

(continued from previous page)

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

Plugins

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*

Language Cache

django-cms caches pages on a per-language basis. Each page version in a different language has its own cache entry, built from:

- Site ID and language
- Page path hash
- Timezone (if USE_TZ is enabled)

The system handles language fallbacks automatically - if a page doesn't exist in the requested language, the fallback language version is cached and served instead.

To manually clear the cache for a specific language:

```
page.clear_cache(language='en')
```

How to enable frontend editing for Page and Django models

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.

> **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"></
↪template>
    <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"></template>
```

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 %}
```

Plugins

Added in version 4.2: Exposing plugins for frontend editing is a new feature in django CMS 4.2.

The frontend editing feature is also available for plugins. While by definition plugins are frontend-editable, you can still use the template tags to expose only selected fields for editing.

Say, you have a header plugin that contains a section header and a summary of the section content. You can expose only the header for editing:

```
{% load cms_tags %}

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

This will render `{{ instance.header }}` and double-clicking on it will open a pop-up window with the change form for the plugin instance with the header field only.

An alternative is to use the `render_model_block` template tag:

```
{% load cms_tags %}

{% block content %}
<h1>{% render_model_block instance "header" %}
  {{ instance.header }}
{% endrender_model_block %}</h1>
{% endblock content %}
```

Third party packages such as djangocms-text use this feature to allow inline editing of single fields in the frontend.

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.

How to manage Page Types

Changed in version 4.0.

Currently, page types as know from django CMS 3.x are not available.

How to run django CMS in headless mode

Added in version 4.2.

Django CMS is headless-ready. This means that you can use django CMS as a backend service to provide content to the frontend technology of your choice.

Traditionally, django CMS serves the content as HTML pages. In headless mode, django CMS does not publish the html page tree. To retrieve content in headless mode you will need an application that serves the content from the CMS via an API, such as djangocms-rest.

Note

When starting a new project with the *djangocms* command, you can scaffold it directly in the mode you want using the `--mode` option:

```
djangocms my_project --mode headless
```

Valid values are `traditional` (the default), `headless` and `hybrid`. The project template uses this to pre-configure the project's `urls.py` and settings accordingly, so the manual steps below are only needed when converting an existing project.

You can also add django CMS to an **existing** project in headless mode by running `djangocms . --mode headless` in the project directory. This adds `djangocms_rest` and `rest_framework` (Django REST framework) to `INSTALLED_APPS`, wires the REST API into `urls.py`, and — because no templates are used — configures *CMS_PLACEHOLDERS* instead of *CMS_TEMPLATES*. See *djangocms* for details.

To run django CMS in headless mode, you simply remove the catch-all URL pattern from your project's `urls.py` file and replace it by an API endpoint:

```
urlpatterns = [
    path('admin/', admin.site.urls),
    # path('', include('cms.urls'))  # Remove this line
]
```

Now, django CMS will be fully accessible through the admin interface, but the frontend will not be served. Once, you add an API endpoint, this will be the only way to access the content.

Note

You can also run a hybrid mode where you serve **both** the HTML pages and the content via an API, say, for an app. In this case, keep the django CMS' URLS and just add the API to your traditional project.

To add an API endpoint, you can use the `djangocms-rest` package, for example. This package provides a REST API for django CMS. To install it, run:

```
pip install djangocms-rest
```

Then, add the following to your `urls.py` file:

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/', include('djangocms_rest.urls')),
]
```

Note

Django CMS does not force you to use the `djangocms-rest` package. You can use any other package that provides an API for django CMS, with a different API such as GraphQL, for example.

If you are using a different API package, you will need to follow the instructions provided by that package.

Headless using templates

In traditional Django CMS, placeholders are defined in the templates and they represent the regions where your plugins (the content) will be rendered. This is easily done via using `{% placeholder "placeholder_name" %}` in your Django templates.

If you keep the `CMS_TEMPLATES` setting in your project, you still will be using templates to render the content when editing and previewing in headless mode. In this case, the templates will be used to identify the placeholders of a page.

This scenario requires templates to be present in the project for the benefit of the editors only.

Headless without templates

However, when running Django CMS headlessly without templates, you fully decouple the front-end presentation layer (which includes templates) from the CMS, and the configuration of placeholders must be handled differently.

First, set the *CMS_TEMPLATES* setting to an empty list in your project's `settings.py` file (or remove it entirely):

```
CMS_TEMPLATES = []
```

Then, you can define the placeholders using the *CMS_PLACEHOLDERS* setting:

```
CMS_PLACEHOLDERS = (
    ('single', ('content',), _('Single placeholder')),
    ('two_column', ('left', 'right'), _('Two columns')),
)
```

The *CMS_PLACEHOLDERS* setting is a list of tuples. Each tuple represents a placeholder configuration. Think of each placeholder configuration replacing a template and providing the information on which placeholders are available on a page: Like a template can have multiple `{% placeholder %}` template tags, a placeholder configuration can contain multiple placeholders.

The first element of the configuration tuple is the name of the placeholder configuration. It is stored in a page's `template` field. It needs to be unique. The second element is a tuple of placeholder slots available for the configuration. The third element is the verbose description of the placeholder configuration which will be shown in the toolbar. You can select a page's placeholder configuration in the Page menu (instead of a template).

Note

CMS_PLACEHOLDERS is only relevant, if no templates are used. If you define templates, placeholders are inferred from the templates.

Also, do not confuse the *CMS_PLACEHOLDERS* setting with the *CMS_PLACEHOLDER_CONF* setting. The latter is used to configure individual placeholders, while the former is used to define available placeholders for a page.

This scenario is useful when you do not want to design templates and focus on the content structure only. Editors will see a generic representation of the plugins in a minimally styled template. Note that the `css` and `js` block of the plugin templates will be loaded also in this case.

Headless setup and app hooks

When running Django CMS in headless mode, you can still use app hooks to integrate your Django apps with the CMS. App hooks allow you to attach Django apps to a CMS page and render the app's content on that page. Those apps will be served via django CMS' url patterns.

If the app provides API endpoints itself, they will need to be included explicitly in the REST API. Please check the package you are using to create the REST API on how to do this.

Hybrid setup

You can also use django CMS in a hybrid setup, where you serve both the HTML pages and the content via an API. In this case, you keep the django CMS' URLS and just add the API to your traditional project.

Be careful, however, to have the API endpoints in your project's urls **before** django CMS' catch-all HTML urls. Otherwise you run the risk of pages with the wrong path shaddowing out the API endpoints.

Creating new functionality

How to create Plugins

The simplest plugin

We'll start with an example of a very simple plugin.

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

Place your plugins in `cms_plugins.py`. 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 gettext_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.gettext_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 -m manage shell
>>> from importlib import import_module
>>> m = import_module("myapp.cms_plugins")
>>> m.some_test_function()  # from the myapp.cms_plugins module
```

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 gettext_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

    def render(self, context, instance, placeholder):
        context = super().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*. While the use of `CMSPlugin.page` is deprecated the property still exists as a compatibility shim.

Handling Relations

Some user interactions make it necessary to create a copy of the plugin, most notably if a user copies and pastes contents of a placeholder. 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)
```

(continues on next page)

(continued from previous page)

```
    sections = models.ManyToManyField(Section)

    def copy_relations(self, oldinstance):
        self.sections.set(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.

Adding a model to an existing custom plugin

When enhancing an existing django CMS plugin with additional functionality, you might need to associate a database model with the plugin. This allows the plugin to store and manage data persistently. However, introducing a model to an existing plugin requires careful handling to prevent the disappearance of existing plugin instances, as discussed in Issue #7476.

1. **Define the Model**:

   In your application's *models.py*, define a model that inherits from *CMSPlugin*. This model will store the plugin's data. To allow for automatic migration later, make sure that all model fields have meaningful defaults.

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

   class MyPluginModel(CMSPlugin):
       title = models.CharField(max_length=100, default='Default Title')  # Add␣
   ↪defaults
       # Add other fields as needed

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

2. **Update the Plugin Class**:

   In your *cms_plugins.py*, associate the plugin with the newly created model by setting the *model* attribute.

   ```
   from cms.plugin_base import CMSPluginBase
   from cms.plugin_pool import plugin_pool
   from django.utils.translation import gettext_lazy as _
   from .models import MyPluginModel

   @plugin_pool.register_plugin
   class MyPlugin(CMSPluginBase):
       model = MyPluginModel
       name = _("My Plugin")
       render_template = "my_app/my_plugin_template.html"
       # Configure other attributes as needed
   ```

3. **Create Migrations**:

Generate the necessary database migrations to reflect the new model. Do not apply them yet, since the migrations will need an additional script to run.

```
python manage.py makemigrations
```

4. **Handle Existing Plugin Instances**:

   After adding a model to an existing plugin, existing instances will not automatically associate with the new model, leading to their disappearance in the CMS interface. To address this, use a migration script to convert existing plugin instances to the new model.

   This script can be added to the migration just created in step 3.

```
def add_model_to_plugin(apps, schema_editor):
    """ Adds instances for the new model.
    ATTENTION: All fields of the model must have a valid default value!"""

    # Adjust the following two lines
    model = app.get_model("my_app", "MyGreatPluginModel")  # Name of the plugin's new␣
↪model class
    plugin_type = "MyGreatPlugin"  # Name of the plugin class

    CMSPlugin = apps.get_model("cms", "CMSPlugin")

    plugin_instances = CMSPlugin.objects.filter(plugin_type=plugin_type)
    for plugin_instance in plugin_instances:
        logger.info('Creating new model instance for plugin instance %s', plugin_
↪instance.pk)
        obj = model()
        obj.pk = plugin_instance.pk
        obj.cmsplugin_ptr = plugin_instance
        obj.plugin_type = plugin_type
        obj.placeholder = plugin_instance.placeholder
        obj.parent = plugin_instance.parent
        obj.language = plugin_instance.language
        obj.position = plugin_instance.position
        obj.creation_date = plugin_instance.creation_date
        obj.save()

class Migration(migrations.Migration):
    ...

    operations = [
        ...,
        migrations.RunPython(add_model_to_plugin),  # Add at the end of migration␣
↪operations
    ]

This script migrates existing plugin instances to the new model structure, preserving␣
↪data integrity.
```

5. **Run the migrations**:

   Apply the modified migration.

```
python manage.py makemigrations
```

After applying migrations, thoroughly test the plugin to ensure that existing instances appear correctly and that the new model functionalities work as intended.

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().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**.

Inline JavaScrip code

django CMS does not enforce a specific way to include JavaScript code in your plugins. Inline JavaScript code is a potential security risk, so it is recommended to avoid it where possible. Since version 4.2 django CMS itself has removed any inline JavaScript from its code base to allow for meaningful Content Security Policy (CSP) headers to be set. **It is good practice to avoid inline JavaScript in your plugins as well.** If your project's CSP policy does not allow inline JavaScript, inline JavaScript provided through Sekizai also will not be executed.

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>{
```

(continues on next page)

(continued from previous page)

```
↪% 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 %}
```

Note

Due to the faster way of updating content in edit mode, <script> tags do not need to be marked with classes like `cms-trigger-event-document-DOMContentLoaded`, `cms-trigger-event-window-DOMContentLoaded`, or `cms-trigger-event-window-load` as in earlier versions of django CMS. Each script in the Sekizai `js` block is now executed in the order they are defined in the template. After all scripts have been loaded and executed, the document's `DOMContentLoaded`, the window's `load` events are triggered. The `cms-content-refresh` jQuery event on the window is also triggered for backwards compatibility.

Some plugins might need to run a certain bit of javascript after a content refresh. This is why after a content refresh in edit mode, the `document.DOMContentLoaded`, `window.load` events and – for backward compatibility – the `cms-content-refresh` jQuery event are triggered. We encourage to use the `window.load` event for your plugin's javascript code that needs to be run after a content refresh. Since this events is triggered both on initial load and on updates (of potentially different plugins), you should make sure that running the event listener multiple times on one plugin should not cause any issues.

```
window.on('load', function () {
    // Here comes your code of the plugin's javascript which
    // needs to be run after a content refresh
});
```

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 tag, you might end up having 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:
```

(continues on next page)

(continued from previous page)

```
        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']
    # Entries may be glob patterns, e.g. child_classes = ['Bootstrap*'].
    # As a special case, child_classes = 'auto' accepts exactly those plugins
    # that name this plugin in their parent_classes (the children opt in).

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

(continues on next page)

(continued from previous page)

```
@plugin_pool.register_plugin
class ChildCMSPlugin(CMSPluginBase):
    render_template = "child.html"
    name = "Child"
    model = ChildPlugin
    # Restricting which plugins are accepted as parents is preferred over
    # setting require_parent = True: naming concrete parent_classes already
    # forces the plugin to have a parent, so the two together are redundant.
    # Here "*" expands to every registered plugin, meaning any plugin is an
    # acceptable parent -- but the plugin must have one (it cannot be added
    # directly to a placeholder).
    parent_classes = ['*']
    # Narrow this down to specific parents whenever you can, e.g.
    # parent_classes = ['ParentCMSPlugin']. Entries may be glob patterns,
    # e.g. parent_classes = ['Bootstrap*'].

    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>
```

If you have attributes of the parent plugin which you need to access in the child you can access the parent instance using get_bound_plugin:

```
class ChildPluginForm(forms.ModelForm):

    class Meta:
        model = ChildPlugin
        exclude = ()

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        if self.instance:
            parent, parent_cls = self.instance.parent.get_bound_plugin()
```

Restricting plugins to specific models

Added in version 5.1.

Django CMS allows you to control which plugins can be used with which models through two complementary filtering mechanisms:

Plugin-level filtering (`allowed_models`)

The `allowed_models` attribute on a plugin class restricts where that plugin can be used. This is useful when you have a plugin that only makes sense in specific contexts.

```
@plugin_pool.register_plugin
class PageSpecificPlugin(CMSPluginBase):
    name = "Page Only Plugin"
    model = CMSPlugin
    render_template = "page_specific.html"
    allowed_models = ["cms.pagecontent"]  # Only available on CMS pages

@plugin_pool.register_plugin
class BlogPlugin(CMSPluginBase):
    name = "Blog Plugin"
    model = CMSPlugin
    render_template = "blog.html"
    allowed_models = ["blog.blogpost", "blog.blogcategory"]
```

The `allowed_models` attribute accepts:

- `None` (default): The plugin is available for all models with placeholders
- A list of model identifiers in the format `"app_label.modelname"` (e.g., `["cms.pagecontent", "myapp.mymodel"]`)
- An empty list `[]`: The plugin cannot be used with any model

> **Note**
>
> Model identifiers are automatically normalized to lowercase, so `["cms.PageContent"]` and `["cms.pagecontent"]` are equivalent.

Use plugin-based restrictions (`allowed_models`) when:

- You have a plugin that only works with specific model types
- You want to prevent misuse of a plugin in inappropriate contexts
- The plugin accesses model-specific attributes or methods (e.g., by evaluating `plugin.placeholder.source`)

Model-level filtering (`allowed_plugins`)

The `allowed_plugins` attribute on a model class restricts which plugins can be added to placeholders on that model. This is useful when you want to limit the available plugins for a specific content type.

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

(continues on next page)

(continued from previous page)

```
class BlogPost(models.Model):
    title = models.CharField(max_length=200)

    placeholders = PlaceholderRelationField()

    # Only allow specific plugins in blog posts
    allowed_plugins = ['TextPlugin', 'LinkPlugin', 'PicturePlugin']

class LandingPage(models.Model):
    title = models.CharField(max_length=200)

    placeholders = PlaceholderRelationField()

    # Restrict to layout and hero plugins
    allowed_plugins = ['MultiColumnPlugin', 'HeroPlugin', 'CallToActionPlugin']
```

The `allowed_plugins` attribute accepts:

- `None` (default): All plugins are allowed (subject to their `allowed_models` filter)
- A list of plugin class names (e.g., `["TextPlugin", "LinkPlugin"]`)
- An empty list `[]`: No plugins are allowed

Use model-level filtering (`allowed_plugins`) when:

- You want to provide a curated editing experience for specific content types
- You need to enforce content guidelines or branding requirements
- You want to simplify the plugin selection for content editors

Combined filtering

When both `allowed_models` (on the plugin) and `allowed_plugins` (on the model) are defined, **both filters must pass** for a plugin to be available:

```
# Plugin definition
@plugin_pool.register_plugin
class ArticlePlugin(CMSPluginBase):
    name = "Article Plugin"
    model = CMSPlugin
    render_template = "article.html"
    allowed_models = ["blog.blogpost", "news.article"]

# Model definition
class BlogPost(models.Model):
    placeholders = PlaceholderRelationField()
    allowed_plugins = ['TextPlugin', 'ArticlePlugin', 'PicturePlugin']
```

In this example:

- `ArticlePlugin` will be available in `BlogPost` (passes both filters)
- `TextPlugin` will be available in `BlogPost` (no model restriction on plugin, listed in model's `allowed_plugins`)
- `ArticlePlugin` will **not** be available in `cms.PageContent` (not in plugin's `allowed_models`)

- `VideoPlugin` will **not** be available in `BlogPost` (not in model's `allowed_plugins`)

Example: Blog with restricted plugins

```
# models.py
class BlogPost(models.Model):
    title = models.CharField(max_length=200)
    author = models.ForeignKey(User, on_delete=models.CASCADE)

    placeholders = PlaceholderRelationField()

    # Main content: rich editing tools
    allowed_plugins = [
        'TextPlugin',
        'PicturePlugin',
        'VideoPlugin',
        'QuotePlugin',
    ]

# cms_plugins.py
@plugin_pool.register_plugin
class QuotePlugin(CMSPluginBase):
    name = "Quote"
    model = QuoteModel
    render_template = "quote.html"
    # Only allow in blog posts and articles
    allowed_models = ["blog.blogpost", "news.article"]
```

Note

If a plugin name in `allowed_plugins` doesn't match any registered plugin, or a model identifier in `allowed_models` doesn't correspond to any installed model, they are silently ignored without raising an error.

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.

For this purpose you can overwrite the two methods on CMSPluginBase.

- *get_extra_placeholder_menu_items()*
- *get_extra_plugin_menu_items()*

Example:

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

(continues on next page)

(continued from previous page)

```
    def render(self, context, instance, placeholder):
        context = super().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_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 = [
            re_path(r'^create_alias/$', self.create_alias, name='cms_create_alias'),
        ]
        return urlpatterns

    def create_alias(self, request):
        if not request.user.is_staff:
            return HttpResponseForbidden("not enough privileges")
        if not 'plugin_id' in request.POST and not 'placeholder_id' in request.POST:
            return HttpResponseBadRequest(
                "plugin_id or placeholder_id POST parameter missing."
```

(continues on next page)

(continued from previous page)

```
            )
        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")
```

Creating and deleting plugin instances

Added in version 4.0.

Plugins live inside placeholders. Since django CMS version 4 placeholders manage the creation, and especially the deletion of plugins. Besides creating (or deleting) database entries for the plugins the placeholders make all necessary changes to the entire plugin tree. **Not using the placeholders to create or delete plugins can lead to corrupted plugin trees.**

- Use *cms.models.placeholdermodel.Placeholder.add_plugin()* or *cms.api.add_plugin()* to create plugins:

```
new_instance = MyPluginModel(
    plugin_data="secret"
```

(continues on next page)

(continued from previous page)

```
    placeholder=placeholder_to_add_to,
    position=1,  # First plugin in placeholder
)

placeholder_to_add_to.add_plugin(new_instance)
assert new_instance_pk is not None  # Saved to db
```

or:

```
new_plugin = cms.api.add_plugin(
    placeholder_to_add_to,
    "MyPlugin",
    position='first-child',  # First position in placeholder (no parent)
    data=dict(plugin_data="secret"),
)
```

- **Use *cms.models.placeholdermodel.Placeholder.delete_plugin()* to delete a plugin including its children**:

```
old_instance.placeholder.delete_plugin(old_instance)
```

Warning

Do not use `PluginModel.objects.create(...)` or `PluginModel.objects.delete()` to create or delete plugin instances. This most likely either throw a database integrity exception or create a inconsistent plugin tree leading to unexpected behavior.

Also, **do not** use `queryset.delete()` to remove multiple plugins at the same time. This will most likely damage the plugin tree.

Marking plugins as slots

Added in version 5.1.

You can mark a plugin as a *slot* — a structural container that is not directly editable by the user — by setting the *is_slot* attribute to `True` on the plugin class. The plugin will still render normally, but double-clicking it in the structure board will not open the edit dialog. Moving the plugin or adding child plugins is not affected.

```
from cms.plugin_base import CMSPluginBase
from cms.plugin_pool import plugin_pool

@plugin_pool.register_plugin
class SeparatorPlugin(CMSPluginBase):
    name = "Separator"
    render_template = "separator.html"
    is_slot = True
```

This is useful for plugins that have no configurable fields or that are fully managed by their parent plugin.

Marking third-party plugins as slots

You can also mark plugins provided by third-party apps as slots without modifying their source code. To do so, set *is_slot* on the plugin class in your app's `AppConfig.ready()` method:

```
from django.apps import AppConfig

class MyAppConfig(AppConfig):
    name = "myapp"

    def ready(self):
        from cms.plugin_pool import plugin_pool

        # Mark a third-party plugin as a slot to simplify
        # the editor experience
        plugin_pool.get_plugin("SomeThirdPartyPlugin").is_slot = True
```

Customising structure board appearance

Added in version 5.1.

You can customise the appearance of a plugin's entry in the structure board by defining an `add_structureboard_classes` method on the plugin's **model** (i.e. the `CMSPlugin` subclass, not the `CMSPluginBase` subclass). If the method exists, its return value is added as CSS classes to the `cms-draggable` container that wraps the plugin in the structure board.

Combined with custom CSS — for example loaded via a custom toolbar or admin stylesheet — this lets you visually distinguish plugins based on their state or configuration.

A typical use case is marking deactivated or draft plugins so editors can spot them at a glance.

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

class AlertPlugin(CMSPlugin):
    message = models.CharField(max_length=200)
    is_active = models.BooleanField(default=True)

    def add_structureboard_classes(self):
        if not self.is_active:
            return "cms-plugin-inactive"
        return ""
```

Then provide matching CSS (e.g. in a custom admin stylesheet):

```
/* Grey out inactive plugins in the structure board */
.cms-draggable.cms-plugin-inactive > .cms-dragitem {
    opacity: 0.45;
}
```

The classes are also applied to the plugin's **children**, so you can style an entire sub-tree. For example, to add a left border to all children of a deactivated plugin:

```
.cms-draggable.cms-plugin-inactive .cms-draggable {
    border-left: 3px solid #c00;
}
```

Note

The method is called during structure board rendering. Keep it fast — avoid database queries. Use values that are already available on the model instance.

How to upgrade custom plugins for django CMS v4+

Difference between django CMS v3 and v4 plugins

The main difference between plugins of django CMS version 3 and django CMS v4 is how the tree is stored in the database. Up to django CMS version 3, the plugin model *CMSPlugin* inherited from a tree model `MP_Node` declared in the django-treebeard library.

As of django CMS version 4, *CMSPlugin* inherits directly from `django.db.models.Model` and manages the tree structure with the two fields *parent* and *position* using SQL Common Table Expressions (CTE) which allow recursive SQL statements. Consequently all model fields originating with treebeard are not available in django CMS v4+.

Warning

Django CMS 4 removes the following fields form *CMSPlugin*:

- `depth`
- `numchild`
- `path`

Also, the meaning of the *position* field has changed. Im django CMS v3 it was unique for each *parent* value (including `None` for plugins at root level). From django CMS v4 on, it is unique for each *placeholder* and *language* entry. Also, positions are counted from 1 to *n* for all *n* plugins of a placeholder language combination. There must not be gaps in the position field (i.e., a missing position value).

Warning

Since the management of the plugin tree happens within the CMS it is important to use the new placeholder API described in the section *Creating and deleting plugin instances* to create and delete plugins.

What to change

The good news is that most custom plugins will *not* require any changes. This is unless they either directly **access one of the django-treebeard fields** or they **create or delete plugins programmatically**.

Replacing access to django-treebeard fields

If your custom plugin accesses django-treebeard field directly, you will have to change your code. How to do this obviously depends on what your code needs to achieve. Here are some examples:

path

To order a queryset of plugins replace `qs.orderby("path")` by `qs.orderby("position")`.

depth

There is no correspondence to the `depth` field. If needed, it has to be computed:

```
@property
def depth(self):
    if self.parent is None:
        return 1
    return self.parent.depth + 1
```

position

Often changes are made at the leaves of the tree. If you happen to know that the parent plugin does not have grant-children, the quick way to get a django CMS 3 position value is:

```
plugin.position - plugin.parent.position if plugin.parent else plugin.position
```

To calculate the `position` field valid for all cases, you can use this code bit:

```
@property
def v3position(self):
    siblings = CMSPlugin.objects.filter(parent=self.parent).orderby("position")
    pos = 1
    for plugin in siblings:
        if plugin == self:
            return pos
        pos += 1
```

Creating or deleting plugins programmatically

To create a plugin, first build an instance, then add it to its placeholder:

```
my_new_plugin = MyPluginModel(parent=None, position=1, my_config="whatever",␣
↪placeholder=my_placholder)
my_placeholder.add_plugin(my_new_plugin)
```

This example puts the plugin at the first position if the placeholder. Those shortcuts might help:

Position	Meaning
`position=parent.position + 1`	First child of `parent`
`position=parent.position + n`	*n* th child of `parent` if parent does **not** have grand-children
`position=placeholder.get_last_plugin_position(langu + 1`	Last plugin in placeholder

Warning

Do **not** use `MyPluginModel.objects.create()`. It will almost certainly throw a database integrity exception.

Creating "universal" plugins

Some packages introduce universal plugins which can be used both on django CMS 3 and django CMS 4 alike. Examples include djangocms-text or djangocms-frontend.

Here is an excerpt from djangocms-text which needs to be able to create and delete child plugins for text fields. It adds private static methods to

```
@staticmethod
def _create_ghost_plugin(placeholder, plugin):
    """CMS version-save function to add a plugin to a placeholder"""
    if hasattr(placeholder, "add_plugin"):  # available as of CMS v4
        placeholder.add_plugin(plugin)
    else:  # CMS < v4
        plugin.save()  # Plugin is created upon save
```

Similarly, it deletes plugins:

```
@staticmethod
def _delete_plugin(plugin):
    """Version-safe plugin delete method"""
    placeholder = plugin.placeholder
    if hasattr(placeholder, 'delete_plugin'):  # since CMS v4
        return placeholder.delete_plugin(plugin)
    else:
        return plugin.delete()
```

Note

Please consider the different counting schemes for the *position* field.

Adapting your test suite

Test suites often create pages, add plugins that are to be tested, and publish the pages. Since publishing in django CMS 4 is not part of the core any more, a way updating the test suites is to add a test fixture to your tests that provide publish and unpublish functionality.

In the tests themselves all `page.publish()` calls then need to be replaced by `self.publis(page)` calls to the fixture.

Here's an example of test fixture (from djangocms-frontend)

```
from packaging.version import Version

from cms import __version__

DJANGO_CMS4 = Version(__version__) >= Version("4")


class TestFixture:
    """Sets up generic setUp and tearDown methods for tests."""
```

(continues on next page)

(continued from previous page)

```
    if DJANGO_CMS4:  # CMS V4
        def _get_version(self, grouper, version_state, language=None):
            language = language or self.language

            from djangocms_versioning.models import Version

            versions = Version.objects.filter_by_grouper(grouper).filter(
                state=version_state
            )
            for version in versions:
                if (
                    hasattr(version.content, "language")
                    and version.content.language == language
                ):
                    return version

        def publish(self, grouper, language=None):
            from djangocms_versioning.constants import DRAFT

            version = self._get_version(grouper, DRAFT, language)
            if version is not None:
                version.publish(self.superuser)

        def unpublish(self, grouper, language=None):
            from djangocms_versioning.constants import PUBLISHED

            version = self._get_version(grouper, PUBLISHED, language)
            if version is not None:
                version.unpublish(self.superuser)

        def create_page(self, title, **kwargs):
            kwargs.setdefault("language", self.language)
            kwargs.setdefault("created_by", self.superuser)
            kwargs.setdefault("in_navigation", True)
            kwargs.setdefault("limit_visibility_in_menu", None)
            kwargs.setdefault("menu_title", title)
            return create_page(title=title, **kwargs)

        def get_placeholders(self, page):
            return page.get_placeholders(self.language)

    else:  # CMS V3
        def publish(self, page, language=None):
            page.publish(language)

        def unpublish(self, page, language=None):
            page.unpublish(language)

        def create_page(self, title, **kwargs):
            kwargs.setdefault("language", self.language)
            kwargs.setdefault("menu_title", title)
```

(continues on next page)

(continued from previous page)

```
            return create_page(title=title, **kwargs)

        def get_placeholders(self, page):
            return page.get_placeholders()
```

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 under that page's URL path will be passed to the attached application's URLconf.

For the conceptual model ("mount points" and what it means for request routing), see *Application hooks ("apphooks")*.

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:

1. Create a `cms_apps.py` file in your application.
2. Add a *CMSApp* sub-class.
3. Register it with the apphook pool.

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

```
from cms.app_base import CMSApp
from cms.apphook_pool import apphook_pool

@apphook_pool.register
class MyApphook(CMSApp):
    app_name = "myapp"  # must match the application namespace
    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
```

Apphooks for namespaced applications

Your application should use namespaced URLs.

In the example above, the application uses the `myapp` namespace. Your `CMSApp` sub-class **must reflect the application's namespace** in the `app_name` attribute.

The application may specify a namespace by supplying an `app_name` in its `urls.py`, or its documentation might advise that when you include its URLs, you do it like this:

```
path("myapp/", include("myapp.urls", app_name="myapp"))
```

If you fail to do 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.

Apphooks for non-namespaced applications

If you are writing apphooks for third-party applications, you may find one that in fact does not have an application namespace for its URLs. Such an application is liable to run into namespace conflicts, and doesn't represent good practice.

However if you *do* encounter such an application, your own apphook for it will need in turn to forgo the `app_name` attribute.

Note that apphooks without `app_name` attributes can be attached only to one page at a time; attempting to apply them a second time will cause an error. Only one instance of these apphooks can exist.

See *Attaching an application multiple times* for more on having multiple apphook instances.

Returning apphook URLs manually

Instead of defining the URL patterns in another file `myapp/urls.py`, it also is possible to return them manually, for example if you need to override the set provided. An example:

```
from django.urls import path
from myapp.views import SomeListView, SomeDetailView


class MyApphook(CMSApp):
    # ...
    def get_urls(self, page=None, language=None, **kwargs):
        return [
            path("<str:slug>/", SomeDetailView.as_view()),
            path("", SomeListView.as_view()),
        ]
```

However, it's much neater to keep them in the application's `urls.py`, where they can easily be reused.

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 you by re-initialising the URL patterns automatically.

Otherwise, you will need to restart the server 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` for the views `index_view` and `archive_view`:

```
urlpatterns = [
    path("archive/", archive_view),
    path("", index_view),
]
```

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

- `index_view` at `/hello/world/`
- `archive_view` at `/hello/world/archive/`

Using placeholders on an apphooked page

Added in version 5.1.

Even though the URL routing is delegated to your application, the *page* still exists and can carry content in its placeholders. The *placeholder* template tag can therefore be used in your apphook's templates.

By default (when the apphook view does not call `request.toolbar.set_object()`), the *placeholder* tag resolves against the current page's *PageContent*, exactly as it would on a regular CMS page — including `inherit` walking up the page tree.

For example, in the template used by your app's view:

```
{% load cms_tags %}
{% block content %}
    {% placeholder "content" %}
    {# application output below #}
{% endblock %}
```

This works because django CMS sets `request.current_page` for apphook requests.

Note

If your apphook view calls `request.toolbar.set_object(some_model_instance)` with an instance of a model that uses *PlaceholderRelationField*, the *placeholder* tag will instead resolve against that object's placeholders (and `inherit` has no effect in this case). Set the toolbar object only when you want editors to work on that custom model rather than on the page.

Telling the structure board which template the root view uses

When a content editor opens the structure board on an apphooked page, django CMS needs to know *which* template the apphook's root view renders, should it be different from the page's template. This is so it can show the placeholders declared in that template. The rendered template is provided by *get_root_template()*.

By default, `get_root_template` inspects the URL pattern matching the empty path inside *get_urls()* and reads its `template_name`. This works automatically for class-based views in either of these forms:

```
# template_name set on the class
class HomeView(TemplateView):
    template_name = "myapp/home.html"

urlpatterns = [path("", HomeView.as_view())]

# or passed as an as_view() keyword argument
urlpatterns = [path("", TemplateView.as_view(template_name="myapp/home.html"))]
```

Inference cannot work in some cases, in which the lookup falls back to `None`:

- function-based views — the template name lives inside the view body, not on the callable
- views that select their template at runtime
- views that return a fully-rendered `HttpResponse` (for example via the `render()` shortcut)

For these cases, override *`get_root_template()`* on the apphook class. The override receives the same `page` and `language` arguments as *`get_urls()`*, so the returned template can vary per page or language:

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

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

    def get_root_template(self, page=None, language=None, **kwargs):
        return "myapp/home.html"
```

Make sure that the template you return here matches the one your root view actually renders, otherwise the placeholders shown in the structure board will diverge from those rendered on the page.

Sub-pages of an apphooked page

Usually you should not add child pages to a page with an apphook. This is because the apphook "swallows" all URLs below that page, handing them over to the attached application.

In the rare occasion that you nevertheless want to add child pages below an apphooked page, then you must add a special URL pattern to route requests back into the CMS.

For example, if you have an apphooked page at `/hello/` and you want to add a CMS page, and optionally its children below that page using the slug `world`, then rewrite the URL patterns from above as:

```
from django.urls import path, re_path
from cms.views import details

def reroute_cms_page(request, path, page=None):
    language = get_language_from_request(request, check_path=True)
    return details(request, f'{page.get_path(language)}/{path}')

class MyApphook(CMSApp):
    # ...
    def get_urls(self, page=None, language=None, **kwargs):
        return [
            path("archive/", archive_view),
            re_path(r"^(?P<path>world/.*)$", reroute_cms_page, {"page": page}),
```

(continues on next page)

(continued from previous page)

```
            path("", index_view),
        ]
```

Here we created a short function-based view named `reroute_cms_page`. It handles the requests which otherwise would be swallowed by the apphook.

A request starting with the URL `/hello/` then is handled by `index_view`, `/hello/archive/` is handled by `archive_view`, and `/hello/world/`, `/hello/world/foo`, etc. are handled by our special view `reroute_cms_page`, routing the request back to the `detail()` view of Django-CMS.

Managing apphooks

Uninstalling an apphook with applied instances

If you remove an apphook class from your system (in effect uninstalling it) 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 subsequently 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 the CMS management command `uninstall apphooks`. For example:

```
python -m manage cms uninstall apphooks MyApphook MyOtherApphook
```

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

```
python -m manage cms list apphooks
```

See the *Management commands reference* for more information.

Adding menus to apphooks

Generally, it is recommended to allow the user to control whether a menu is attached to a page (See *Attach Menus* for more on these menus). However, an apphook can be made to do this automatically if required. It will behave just as if the menu had been attached to 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.cms_menus 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 has been 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. *Attach Menus* has more information 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]
```

Managing permissions on apphooks

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 MyApphook(CMSApp):
    [...]
    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 make your own permission checks in your application, then use the `exclude_permissions` property of the apphook:

```
class MyApphook(CMSApp):
    [...]
    permissions = True
    exclude_permissions = ["some_nested_app"]
```

where you provide the name of the application in question

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 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.

The signal is fired **after** a request - for example, upon saving a page's settings. If you change an apphook's setting via an API the signal will not fire until a subsequent request.

How to manage complex apphook configurations

This how-to builds on *How to create apphooks* and focuses on two advanced, practical needs:

- attaching the *same* application to multiple CMS pages (multiple mount points)
- selecting per-mount behaviour using editor-managed configuration ("apphook configurations")

Use this guide when you need multiple instances of one apphook (for example `/faq` and `/support/faq`) or when your application must behave differently depending on where it is mounted. Before starting, you should be familiar with the *Application hooks ("apphooks")* conceptual model and have successfully created a basic apphook (see *How to create apphooks*). Understanding Reversing namespaced URLs will also help.

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 will 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 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 determine 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.

Create the new FAQ application

Let us quickly create the new app:

1. **Create a new app in your project**:

```
python -m manage startapp faq
```

2. **Create a model for the app config in ``models.py``**: The app config will be identified by its namespace.

```
from django.db import models
from django.utils.translation import gettext_lazy as _


class FaqConfigModel(models.Model):
    namespace = models.CharField(
        _("instance namespace"),
        default=None,
        max_length=100,
        unique=True,
    )

    paginate_by = models.PositiveIntegerField(
        _("paginate size"),
        blank=False,
        default=5,
    )
```

3. **Create the FAQ model also in `models.py`: All entries will be assigned to an** instance of the apphook.

```
class Entry(models.Model):
    app_config = models.ForeignKey(
        FaqConfigModel,
        null=False,
        on_delete=models.PROTECT,
    )
    question = models.TextField(blank=True, default='')
    answer = models.TextField()

    def __str__(self):
        return self.question or "<Empty question>"

    class Meta:
        verbose_name_plural = 'entries'
```

4. **Create the FAQ CMS app: In the app's `cms_apps.py` create the `FaqConfig`** class. This extension tells django CMS how to get the app config instances.

```
from django.core.exceptions import ObjectDoesNotExist
from django.urls import reverse

from cms.app_base import CMSApp
from cms.apphook_pool import apphook_pool

from .models import FaqConfigModel


@apphook_pool.register
class FaqConfig(CMSApp):
    name = "FAQ"
    app_name = "faq"
    app_config = FaqConfigModel
```

(continues on next page)

(continued from previous page)

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

    def get_configs(self):
        return self.app_config.objects.all()

    def get_config(self, namespace):
        try:
            return self.app_config.objects.get(namespace=namespace)
        except ObjectDoesNotExist:
            return None

    def get_config_add_url(self):
        try:
            return reverse(
                "admin:{}_{}_add".format(
                    self.app_config._meta.app_label,
                    self.app_config._meta.model_name,
                )
            )
        except AttributeError:
            return reverse(
                "admin:{}_{}_add".format(self.app_config._meta.app_label, self.
↪app_config._meta.module_name)
            )
```

5. **Add models to the admin interface**: Its admin properties are defined in `admin.py`:

```
from django.contrib import admin

from . import models


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


@admin.register(models.FaqConfigModel)
class FaqConfigAdmin(admin.ModelAdmin):
    pass
```

6. **Create a simple list view** in `views.py`: For the views there is a catch: The view will have to determine which app instance it is showing. Here's a short reusable mixin to help with that:

```
from django.views.generic import ListView
from django.urls import Resolver404, resolve
from django.utils.translation import override

from cms.apphook_pool import apphook_pool
from cms.utils import get_language_from_request

from . import models


def get_app_instance(request):
    namespace, config = "", None
    if getattr(request, "current_page", None) and request.current_page.
↪application_urls:
        app = apphook_pool.get_apphook(request.current_page.application_
↪urls)
        if app and app.app_config:
            try:
                with override(get_language_from_request(request)):
                    if hasattr(request, "toolbar") and hasattr(request.
↪toolbar, "request_path"):
                        path = request.toolbar.request_path  # If using the␣
↪v4 endpoint, take request_path from toolbar
                    else:
                        path = request.path_info
                    namespace = resolve(path).namespace
                    config = app.get_config(namespace)
            except Resolver404:
                pass
    return namespace, config


class AppHookConfigMixin:
    def dispatch(self, request, *args, **kwargs):
        # get namespace and config
        self.namespace, self.config = get_app_instance(request)
        request.current_app = self.namespace
        return super().dispatch(request, *args, **kwargs)

    def get_queryset(self):
        qs = super().get_queryset()
        return qs.filter(app_config__namespace=self.namespace)


class IndexView(AppHookConfigMixin, ListView):
    model = models.Entry
    template_name = 'faq/index.html'

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

7. **Declare the app's URLs** in `urls.py`:

```
from django.urls import path
from . import views


urlpatterns = [
    path("", views.IndexView.as_view(), name='index'),
]
```

8. Finally, **create a template for the index view**:

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

{% block content %}
    <h1>Namespace: {{ view.namespace }}</h1>
    <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 %}
```

Put it all together

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

```
python -m manage makemigrations faq
python -m manage migrate faq
```

Now we should be all set.

Create two pages with the faq apphook, with different namespaces and different configurations. Also create some entries assigned to the two namespaces.

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 extend the Toolbar

The django CMS toolbar provides an API that allows you to add, remove and manipulate toolbar items in your own code. It helps you to integrate django CMS's frontend editing mode into your application, and provide your users with a streamlined editing experience.

See also

- Extending the Toolbar in the tutorial
- *Toolbar API reference*

Create a `cms_toolbars.py` file

In order to interact with the toolbar API, you need to create a *`CMSToolbar`* sub-class in your own code, and register it.

This class should be created in your application's `cms_toolbars.py` file, where it will be discovered automatically when the Django runserver starts.

You can also use the *`CMS_TOOLBARS`* to control which toolbar classes are loaded.

Use the high-level toolbar APIs

You will find a `toolbar` object in the request in your views, and you may be tempted to do things with it, like:

```
toolbar = request.toolbar
toolbar.add_modal_button('Do not touch', dangerous_button_url)
```

- but you should not, in the same way that it is not recommended to poke tweezers into electrical sockets just because you can.

Instead, you should **only** interact with the toolbar using a `CMSToolbar` class, and the *documented APIs for managing it*.

Similarly, although a generic *`add_item()`* method is available, we provide higher-level methods for handling specific item types, and it is always recommended that you use these instead.

Define and register a `CMSToolbar` sub-class

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

class MyToolbarClass(CMSToolbar):
    [...]

toolbar_pool.register(MyToolbarClass)
```

The `cms.toolbar_pool.ToolbarPool.register` method can also be used as a decorator:

```
@toolbar_pool.register
class MyToolbarClass(CMSToolbar):
    [...]
```

Populate the toolbar

Two methods are available to control what will appear in the django CMS toolbar:

- `populate()`, which is called *before* the rest of the page is rendered
- `post_template_populate()`, which is called *after* the page's template is rendered

The latter method allows you to manage the toolbar based on the contents of the page, such as the state of plugins or placeholders, but unless you need to do this, you should opt for the more simple `populate()` method.

```
class MyToolbar(CMSToolbar):

    def populate(self):

        # add items to the toolbar
```

Now you have to decide exactly what items will appear in your toolbar. These can include:

- *menus*
- *buttons* and button lists
- various other toolbar items

Add links and buttons to the toolbar

You can add links and buttons as entries to a menu instance, using the various `add_` methods.

Action	Text link variant	Button variant
Open link	*add_link_item()*	*add_button()*
Open link in sideframe	*add_sideframe_item()*	*add_sideframe_button()*
Open link in modal	*add_modal_item()*	*add_modal_button()*
Ajax POST action	*add_ajax_item()*	

The basic form for using any of these is:

```
def populate(self):

    self.toolbar.add_link_item( # or add_button(), add_modal_item(), etc
        name='A link',
        url=url
        )
```

Note that although these toolbar items may take various positional arguments in their methods, **we strongly recommend using named arguments**, as above. This will help ensure that your own toolbar classes and methods survive upgrades. See the reference documentation linked to in the table above for details of the signature of each method.

Opening a URL in an iframe

A common case is to provide a URL that opens in a sideframe or modal dialog on the same page. *Administration…* in the site menu, that opens the Django admin in a sideframe, is a good example of this. Both the sideframe and modal are HTML iframes.

A typical use for a sideframe is to display an admin list (similar to that used in the tutorial example):

```
from cms.utils.urlutils import admin_reverse
[...]

class PollToolbar(CMSToolbar):

    def populate(self):

        self.toolbar.add_sideframe_item(
            name='Poll list',
            url=admin_reverse('polls_poll_changelist')
            )
```

A typical use for a modal item is to display the admin for a model instance:

```
self.toolbar.add_modal_item(name='Add new poll', url=admin_reverse('polls_poll_add'))
```

However, you are not restricted to these examples, and you may open any suitable resource inside the modal or sideframe. Note that protocols may need to match and the requested resource must allow it.

Triggering an AJAX action

Use *add_ajax_item()* for an item that performs an action on the server rather than navigating to a page. It sends a request (`POST` by default) to `action`, optionally asking the user to confirm first:

```
from cms.utils.urlutils import admin_reverse

class PollToolbar(CMSToolbar):

    def populate(self):

        self.toolbar.add_ajax_item(
            name='Reset votes',
            action=admin_reverse('polls_poll_reset', args=[self.poll.pk]),
            data={'confirm': True},
            question='Reset all votes for this poll?',
            on_success=self.toolbar.REFRESH_PAGE,
        )
```

The `data` dictionary is sent with the request; the CSRF token is added automatically for unsafe HTTP methods. Use `on_success` to control what happens after a successful response: pass a URL to navigate to it, `REFRESH_PAGE` to reload the current page, or `"FOLLOW_REDIRECT"` to redirect to the `url` returned in a JSON response.

Added in version 5.1: If `on_success` is not given and the response carries a *data bridge* – either an HTML document containing a `<script id="data-bridge">` element or a JSON object with an `action` key – the data bridge is evaluated and the structure board is updated in place, without a full page reload. This is the same mechanism used when a plugin is saved in a modal.

Adding buttons to the toolbar

A button is a sub-class of *cms.toolbar.items.Button*

Buttons can also be added in a list - a *ButtonList* is a group of visually-linked buttons.

```
def populate(self):

    button_list = self.toolbar.add_button_list()
    button_list.add_button(name='Button 1', url=url_1)
    button_list.add_button(name='Button 2', url=url_2)
```

Create a toolbar menu

The text link items described above can also be added as nodes to menus in the toolbar.

A menu is an instance of *cms.toolbar.items.Menu*. In your `CMSToolbar` sub-class, you can either create a menu, or identify one that already exists (in order to add new items to it, for example), in the `populate()` or `post_template_populate()` methods, using *get_or_create_menu()*.

```
def populate(self):
    menu = self.toolbar.get_or_create_menu(
        key='polls_cms_integration',
        verbose_name='Polls'
        )
```

The `key` is unique menu identifier; `verbose_name` is what will be displayed in the menu. If you know a menu already exists, you can obtain it with *get_menu()*.

Note

It's recommended to namespace your `key` with the application name. Otherwise, another application could unexpectedly interfere with your menu.

Once you have your menu, you can add items to it in much the same way that you add them to the toolbar. For example:

```
def populate(self):
    menu = [...]

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

To add a menu divider

add_break() will place a *Break*, a visual divider, in a menu list, to allow grouping of items. For example:

```
menu.add_break(identifier='settings_section')
```

To add a sub-menu

A sub-menu is a menu that belongs to another `Menu`:

```
def populate(self):
    menu = [...]

    submenu = menu.get_or_create_menu(
        key='sub_menu_key',
        verbose_name='My sub-menu'
        )
```

You can then add items to the sub-menu in the same way as in the examples above. Note that a sub-menu is an instance of *SubMenu*, and may not itself have further sub-menus.

Finding existing toolbar items

`get_or_create_menu()` and `get_menu()`

A number of methods and useful constants exist to get hold of and manipulate existing toolbar items. For example, to find (using `get_menu()`) and rename the *Site* menu:

```
from cms.cms_toolbars import ADMIN_MENU_IDENTIFIER

class ManipulativeToolbar(CMSToolbar):

    def populate(self):

        admin_menu = self.toolbar.get_menu(ADMIN_MENU_IDENTIFIER)

        admin_menu.name = "Site"
```

`get_or_create_menu()` will equally well find the same menu, and also has the advantages that:

- it can update the item's attributes itself (`self.toolbar.get_or_create_menu(ADMIN_MENU_IDENTIFIER, 'Site')`)
- if the item doesn't exist, it will create it rather than raising an error.

`find_items()` and `find_first()`

Search for items by their type:

```
def populate(self):

    self.toolbar.find_items(item_type=LinkItem)
```

will find all `LinkItems` in the toolbar (but not for example in the menus in the toolbar - it doesn't search *other* items in the toolbar for items of their own).

find_items() returns a list of *ItemSearchResult* objects; *find_first()* returns the first object in that list. They share similar behaviour so the examples here will use `find_items()` only.

The `item_type` argument is always required, but you can refine the search by using their other attributes, for example:

```
self.toolbar.find_items(Menu, disabled=True))
```

Note that you can use these two methods to search `Menu` and `SubMenu` classes for items too.

Control the position of items in the toolbar

Methods to add menu items to the toolbar take an optional *position* argument, that can be used to control where the item will be inserted.

By default (`position=None`) the item will be inserted after existing items in the same level of the hierarchy (a new sub-menu will become the last sub-menu of the menu, a new menu will be become the last menu in the toolbar, and so on).

A position of `0` will insert the item before all the others.

If you already have an object, you can use that as a reference too. For example:

```
def populate(self):

    link = self.toolbar.add_link_item('Link', url=link_url)
    self.toolbar.add_button('Button', url=button_url, position=link)
```

will add the new button before the link item.

Finally, you can use a *ItemSearchResult* as a position:

```
def populate(self):

    self.toolbar.add_link_item('Link', url=link_url)

    link = self.toolbar.find_first(LinkItem)

    self.toolbar.add_button('Button', url=button_url, position=link)
```

and since the `ItemSearchResult` can be cast to an integer, you could even do:

```
self.toolbar.add_button('Button', url=button_url, position=link+1)
```

Control how and when the toolbar appears

By default, your *CMSToolbar* sub-class will be active (i.e. its `populate` methods will be called) in the toolbar on every page, when the user `is_staff`. Sometimes however a `CMSToolbar` sub-class should only populate the toolbar when visiting pages associated with a particular application.

A `CMSToolbar` sub-class has a useful attribute that can help determine whether a toolbar should be activated. `is_current_app` is `True` when the application containing the toolbar class matches the application handling the request.

This allows you to activate it selectively, for example:

```
def populate(self):

    if not self.is_current_app:
        return

    [...]
```

If your toolbar class is in another application than the one you want it to be active for, you can list any applications it should support when you create the class:

```
supported_apps = ['some_app']
```

`supported_apps` is a tuple of application dotted paths (e.g: `supported_apps = ('whatever.path.app', 'another.path.app')`.

The attribute `app_path` will contain the name of the application handling the current request - if `app_path` is in `supported_apps`, then `is_current_app` will be `True`.

Modifying an existing toolbar

If you need to modify an existing toolbar (say to change an attribute or the behaviour of a method) you can do this by creating a sub-class of it that implements the required changes, and registering that instead of the original.

The original can be unregistered using `toolbar_pool.unregister()`, as in the example below. Alternatively if you originally invoked the toolbar class using *CMS_TOOLBARS*, you will need to modify that to refer to the new one instead.

An example, in which we unregister the original and register our own:

```
from cms.toolbar_pool import toolbar_pool
from third_party_app.cms_toolbar import ThirdPartyToolbar

@toolbar_pool.register
class MyBarToolbar(ThirdPartyToolbar):
    [...]

toolbar_pool.unregister(ThirdPartyToolbar)
```

Detecting URL changes to an object

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:

- in *draft* mode the `get_draft_url()` is returned (or `get_absolute_url()` if the former does not exist)
- in *live* mode, and the method exists, `get_public_url()` is returned.

Frontend

If you need to interact with the toolbar, or otherwise account for it in your site's frontend code, it provides CSS and JavaScript hooks for you to use.

It will add various classes to the page's <html> element:

- `cms-ready`, when the toolbar is ready
- `cms-toolbar-expanded`, when the toolbar is fully expanded

- `cms-toolbar-expanding` and `cms-toolbar-collapsing` during toolbar animation.

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 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 gettext_lazy as _

class TestMenu(Menu):

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

menu_pool.register_menu(TestMenu)
```

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 gettext_lazy as _
from cms.menu_bases import CMSAttachMenu

class TestMenu(CMSAttachMenu):

    name = _("test menu")
```

(continues on next page)

(continued from previous page)

```
    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):
    """
```

(continues on next page)

(continued from previous page)

```
    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):
    """
    marks all node levels
    """
    post_cut = True

    def modify(self, request, nodes, namespace, root_id, post_cut, breadcrumb):
        if breadcrumb:
            return nodes
```

(continues on next page)

(continued from previous page)

```
        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.

Added 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 such as creating a new page.

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
```

(continues on next page)

(continued from previous page)

```
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
```

Then, simply subclass `Wizard` and instantiate it.

Note

Added in version 4.0.

Registering a wizard with the wizard_pool is no longer the preferred way to register a wizard. Since django CMS version 4 django CMS keeps track of wizard using `cms_config.py`.

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()


# 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.

Added in version 4.1.

How to create an admin class for a grouper model

What is a grouper model?

It's an reusable abstract structural pattern, that is in django CMS used to separate language independent and language specific content.

django CMS defines grouper-content structure for Page-PageContent as follows:

- The *Page* is the grouper model which represents base unit, that can have multiple content objects attached
- The *PageContent* is the content model which represents page content that can be different for its grouping field - `language` in our case. It also includes the placeholders for the frontend editor.

This mechanism ensures that language-independent properties of a page, such as position in the page tree or permissions, are collected at the grouper model while language-specific content is collected in the content model.

> **Note**
>
> This pattern is relevant for django CMS Versioning since it versions the content objects and not the grouper objects.
>
> To this end, if you want to create models that should be versionable like the *PageContent* of a *Page* objects you need to define a grouper and a content model.

Extra grouping fields define fields of the content model by which they are grouped: *PageContent* uses `language` as an extra grouping field. This means that one *Page* object can have multiple *PageContent* objects assign to which differ in their language.

If not extra grouping fields are given each grouper object can have at most one content object assigmed to it.

The language field is a typical (but not necessary) extra grouping field.

Administrating grouper models

To simplify creation of grouper content models, django CMS provides support for both the model admin class of the grouper model and the change and add forms of the content model.

In this scenario you will register a model admin for the grouper model and it will provide the user with the ability to view, change and add content objects, too. You will not necessarily need to add a model admin class for the content model at all (with the possible exception of a redirecting stub to allow third party apps to reverse admin views for the content model, too, see below).

To create a model admin class for a grouper model put the following code in your *admin.py*:

```
from cms.admin.utils import GrouperModelAdmin


class MyGrouperAdmin(GrouperModelAdmin):
    # Declare content model
    content_model = MyContent
    # Add language tabs to change and add views
    extra_grouping_fields = ("language",)
    # Add grouper and content fields to change list view
    # Add preview and settings action to change list view
    list_display = (
```

(continues on next page)

(continued from previous page)

```
        "field_in_grouper_model",
        "content__field_in_content_model",
        "admin_list_actions",
    )
```

The property *content_model* defines which model is used as the content model. If you do not specify a *content_model*, django CMS will look for a model named like the grouper model but with "Content" appended. The default content model for `Post` would be `PostContent`.

The content model needs to have a foreign key pointing to the grouper model. The first foreign key found is assumed to be the field by which the content objects are assigned to their grouper objects. If you have multiple foreign keys to the grouper model, please specify *content_related_field*.

For this example there is only `language` as extra grouping field declared. You only have to proviude tuple of *extra_grouping_fields* if you have any.

Note

All fields serving as extra grouping fields must be part of the admin's fieldsets setting for *GrouperModelAdmin* to work properly. In the change form the fields will be invisible.

Change list view

For the list display *GrouperModelAdmin* provides additional fields from the content model: `content__{content_model_field_name}`. Those fields can be used in list_display just as grouper model fields and will automatically show the content of the currently selected grouping fields.

Finally, *GrouperModelAdmin* provides two action buttons for each entry in the change list view:

- to preview the content model in the frontend editor
- to change the settings (i.e., go to the change view of the grouper object)

These are for convenience and appear as soon as `admin_list_actions` is added to the `list_display` attribute.

Example

This is an example (taken from django CMS alias) on how a grouper admin might look like:

```
from cms.admin.utils import GrouperModelAdmin


@admin.register(Alias)
class AliasAdmin(GrouperModelAdmin):
    list_display = ["content__name", "category", "admin_list_actions"]
    list_display_links = None  # With action buttons a link is not needed
    list_filter = (
        SiteFilter,
        CategoryFilter,
    )  # Custom filters
    fields = (
        "content__name",
        "category",
        "site",
```

(continues on next page)

(continued from previous page)

```
        "content__language",
    )  # feeds into fieldsets
    readonly_fields = ("static_code",)
    form = AliasGrouperAdminForm  # Custom admin form
    extra_grouping_fields = ("language",)  # Language as grouping field
    EMPTY_CONTENT_VALUE = mark_safe(
        _("<i>Missing language</i>")
    )  # Label for missing content objects
```

Other extra grouping fields (besides language)

The standard templates of django CMS will work with `language` as an extra grouping field out of the box:

- It creates a dropdown to switch languages for the admin's change list view.
- It creates tabs to switch languages for the admin's change and add views.

To use other grouping fields you will have to do two things:

1. You will need to **supply templates** for the change list view and the change and add views that render corresponding dropdowns or other ways of selecting which content is currently being viewed.
2. You will need to **provide context** for the templates to render the valid choices.

Providing your own templates

To show a selector for your additional grouping field you need to overwrite both the `change_list_template` and `change_form_template`. Your templates can extend the default templates. Let's say you have "region" as an additional grouping field. For the **change list template** this might look like this:

```
{% extends "admin/cms/grouper/change_list.html" %}
{% block language_tabs %}
    {# Here goes the region mark-up #}
    {% if region_dropdown %}
        <div class="region-selector">
            ...
        </div>
    {% endif %}
    {{ block.super }}
{% endblock %}
```

For the **change form template** this might look like this:

```
{% extends "admin/cms/grouper/change_form.html" %}
{% block search %}
    {# Here goes the region mark-up #}
    {% if "region" in cl.model_admin.extra_grouping_fields %}
        <div class="region-selector">
            ...
        </div>
    {% endif %}
    {{ block.super }}
{% endblock %}
```

Providing the required context

To provide the required context for your additional grouping model, you will have to implement two methods in your grouper model admin.

```
from cms.admin.utils import GrouperModelAdmin


class MyGrouperAdmin(GrouperModelAdmin):
    model = MyModel
    extra_grouping_fields = ("region",)

    ...

    def changelist_view(request, extra_context=None):
        """Extra context for changelist_view"""
        my_context = {...}  # Add context on region grouper
        return super().changelist_view(
            request, extra_context={**(extra_context or {}), **my_context}
        )

    def get_extra_context(self, request, obj_id=None):
        """Extra context for add_view and change_view"""
        my_context = {...}  # Add context on region grouper
        return {
            **super().get_extra_context(request, obj_id),
            **my_context,
        }
```

Consider that the context will require a set of values your additional grouping field can take. In the region example this might be `all_regions = {"americas": _("Americas"), "europe": _("Europe"), ...}`.

How to extend Page & PageContent models

You can extend the *cms.models.pagemodel.Page* and *cms.models.contentmodels.PageContent* models with your own fields (e.g. adding an icon for every page) by using the extension models: `cms.extensions.PageExtension` and `cms.extensions.PageContentExtension`, respectively.

Note

Changed in version 4.1: In django CMS the `PageContent` model used to be called `Title`. Since django CMS 4.1 a `TitleExtension` has become `PageContentExtension`

PageContent vs Page extensions

The difference between a **page extension** and a **page content extension** is related to the difference between the *cms.models.pagemodel.Page* and *cms.models.contentmodels.PageContent* models.

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

Continuing with the example model above, here's a simple corresponding `PageExtensionAdmin` class that should live in the `admin.py` file:

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

from .models import IconExtension


class IconExtensionAdmin(PageExtensionAdmin):
```

(continues on next page)

(continued from previous page)

```
    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 gettext_lazy as _
from .models import IconExtension


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

    def populate(self):
        # setup the extension toolbar with permissions and sanity checks
        current_page_menu = self._setup_extension_toolbar()

        # if it's all ok
        if current_page_menu:
            # retrieves the instance of the current extension (if any) and the toolbar␣
↪item URL
            page_extension, url = self.get_page_extension_admin()
            if url:
                # adds a toolbar item 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_active, position=0)
```

PageContent model extension example

In this example, we'll create a `Rating` extension field, that can be applied to each `PageContent`, 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 PageContentExtension
from cms.extensions.extension_pool import extension_pool


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


extension_pool.register(RatingExtension)
```

The admin

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


class RatingExtensionAdmin(PageContentExtensionAdmin):
    pass


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

The toolbar item

In this example, we need to loop over the page contents 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 gettext_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):
```

(continues on next page)

(continued from previous page)

```
        # 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_active:
            # 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
                )

            # we now need to get the pagecontent_set (i.e. different language page␣
↪contents)
            # for this page
            page = self._get_page()
            page_contents = page.pagecontent_set(manager="admin_manager").latest_
↪content(language__in=get_language_list(page.node.site_id))

            # create a 3-tuple of (title_extension, url, title)
            nodes = [
                (*self.get_page_content_extension_admin(page_content), page_content.
↪title)
                for page_content in page_contents
            ]

            # 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_
↪active
                    )
```

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 static %}

{# rest of template omitted ... #}

{% if request.current_page.iconextension %}
```

(continues on next page)

(continued from previous page)

```
    <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.

PageContent extensions

In order to retrieve a page content extension within a template, get the `PageContent` object using `request.current_page.get_content_obj`. Using the example above, we could use:

```
{{ request.current_page.get_content_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 `PageContentExtension` 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.urls import reverse, NoReverseMatch
from django.utils.translation import gettext_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
            except NoReverseMatch:
                # not in urls
                pass
            else:
                not_edit_mode = not self.toolbar.edit_mode_active
                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:

- `ExtensionToolbar.get_page_extension_admin()`: for page extensions, retrieves the correct admin URL for the related toolbar item; returns the extension instance (or `None` if none exists) and the admin URL for the toolbar item
- `ExtensionToolbar.get_page_content_extension_admin(page_content=None)`: for page content extensions, retrieves the correct admin URL for the related toolbar item; returns a tuple of the extension instance (or `None` if none exists) and the admin URL for the current page content (if the argument is None or omitted) or the page content object passed.

  Typically, `ExtensionToolbar.get_page_content_extension_admin` is used without the argument to modify the toolbar for the currently visible page content object.

> **Warning**
>
> The `ExtensionToolbar.get_title_extension_admin(language=None)` from django CMS versions before 4.1 still exists but is deprecated.

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.urls import re_path, include

urlpatterns = [
    re_path(r'^admin/', admin.site.urls),
    re_path(r'^myapp/', include('myapp.urls')),
    re_path(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
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>')
```

Sharing functionality

How to share capabilities between apps

Added in version 4.0.

To understand how to use the app registration system, lets use an example. Let's say our `INSTALLED_APPS` include these three apps:

```
INSTALLED_APPS = [
    ...
    'pink_cms_admin',
    'pony_cms_icons',
    'blog_posts',
]
```

The `pink_cms_admin` is an app that extends the cms by making apps, that are accordingly configured, to have a pink admin. To do that, it would define a `pink_cms_admin/cms_config.py` file, which would look like this:

```
from cms.app_base import CMSAppExtension

from pink_cms_admin import make_admin_pink


class PinkAdminCMSExtension(CMSAppExtension):

    def configure_app(self, cms_config):
         # Do anything you need to do to each app that wants to be pink
         make_admin_pink(cms_config)
```

The `blog_posts` app wants to be pink and wants to have pony icons everywhere. So it would define `blog_posts/cms_config.py` like this:

```
from cms.app_base import CMSAppConfig


class BlogPostsCMSConfig(CMSAppConfig):
    # To enable functionality define an attribute like <app_label>_enabled
    # and set it to True
    pink_cms_admin_enabled = True
    pony_cms_icons_enabled = True

    # pony_cms_icons also has additional settings. These are defined here.
    pony_cms_icons_pony_colours = ['purple', 'pink']
    pony_cms_icons_ponies_with_wings = True
```

The pony_cms_icons app lets other apps have pony icons everywhere, but also wants to have a pink admin. So it would define `pony_cms_icons/cms_config.py` like this:

```
from django.core.exceptions import ImproperlyConfigured

from cms.app_base import CMSAppConfig, CMSAppExtension

from pony_cms_icons import add_pony_icons

```

(continues on next page)

(continued from previous page)

```
class PonyIconsCMSConfig(CMSAppConfig):
    pink_cms_admin_enabled = True


class PonyIconsCMSExtension(CMSAppExtension):

    def configure_app(self, cms_config):
        # Do anything you need to do to each app that wants to have
        # pony icons here

        # As pony icons defines additional settings, you will also need to check
        # for any required settings here
        pony_colours = getattr(cms_config, 'pony_cms_icons_pony_colours', None)
        if not pony_colours:
            raise ImproperlyConfigured(
                "Apps that use pony_cms_icons, must define pony_cms_icons_pony_colours")
        ponies_with_wings = getattr(cms_config, 'pony_cms_icons_ponies_with_wings',␣
↪False)

        add_pony_icons(cms_config.django_app, pony_colours, ponies_with_wings)
```

The *configure_app()* method, as is already apparent, takes one param - `cms_config`. `cms_config` is an instance of an app's *CMSAppConfig* class. In addition to that you can also access the django app object (as defined in the app's apps.py) by using `cms_config.app_config`. In this way you can access attributes that django provides (such as `label`, `verbose_name` etc.).

The *configure_app()* method is run once for every django cms app that declares a feature as enabled.

If an app asks for a feature of another app that is not installed this feature is simply ignored. This in turn implies that you cannot assume that the feature you request in a *CMSAppConfig* is also available. Therefore, make sure your app's code also runs without that feature or check if your providing app is present in your *CMSAppConfig* and raise an `ImproperlyConfigured` exception if it is missing.

5.3.4 Reference

Technical reference material.

Command Line Interface

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

```
python manage.py cms
```

Creating a project

djangocms

The `djangocms` command creates a new django CMS project directory structure for the given project name. It is available as a standalone executable once django CMS is installed and can also be invoked as `python -m cms`:

```
djangocms <project_name> [directory]
```

It clones the official cms-template matching your installed django CMS version, installs the project requirements, runs the initial migrations, creates a superuser and checks the installation.

It accepts the following options:

- `--interactive`: ask for the project name and for any option not given on the command line. Interactive mode is also entered automatically when no project name is given (unless `--noinput` is used).
- `--noinput` / `--no-input`: do not prompt for any input. A superuser created this way will not be able to log in until given a valid password.
- `--username`: the login for the superuser to be created.
- `--email`: the email for the superuser to be created.
- `--mode {traditional,headless,hybrid}`: select the CMS mode (see *How to run django CMS in headless mode*). Defaults to `traditional`.
  - `traditional`: django CMS serves the content as HTML pages.
  - `headless`: django CMS serves the content only through an API; the HTML page tree is not published.
  - `hybrid`: django CMS serves **both** HTML pages and the content through an API.
- `--versioning` / `--no-versioning`: add content versioning (djangocms-versioning) to the project. On by default.
- `--moderation` / `--no-moderation`: add content moderation (djangocms-moderation) to the project. Off by default. Moderation builds on top of versioning, so it cannot be combined with `--no-versioning`.
- `--alias` / `--no-alias`: add reusable aliases (djangocms-alias) to the project. On by default.
- `--stories` / `--no-stories`: add the stories component library (djangocms-stories) to the project. Off by default.

Example:

```
djangocms my_project --mode headless --no-versioning --stories
```

To be guided through the available options interactively, run the command without a project name (or pass `--interactive`):

```
djangocms
```

Adding django CMS to an existing project

If you pass `.` as the project name, the command does **not** clone the project template. Instead it adds django CMS to the existing Django project in the current directory (see *Add django CMS to an existing project* for a step-by-step guide):

```
djangocms . --mode hybrid --no-versioning
```

The project's settings module is read from `manage.py`, and the following files are updated:

- the settings module (`DJANGO_SETTINGS_MODULE`): the django CMS apps are added to `INSTALLED_APPS` — the core apps plus those selected by the flags (for example `--versioning` adds `djangocms_versioning`). In `headless` and `hybrid` mode the REST API apps `djangocms_rest` and `rest_framework` (Django REST framework) are added as well. The django CMS middleware (including Django's `LocaleMiddleware`) and template context processors are added, and a small block of django CMS settings (such as `LANGUAGES`) is appended if missing. In `traditional` and `hybrid` mode this includes `CMS_TEMPLATES`; in `headless` mode it includes `CMS_PLACEHOLDERS` instead (see *How to run django CMS in headless mode*);

- the project's `urls.py` (resolved from `ROOT_URLCONF`): the django CMS and/or REST API url patterns are added according to the selected `--mode`.

In `traditional` and `hybrid` mode, if the project has no template directory yet, a `templates` directory is created, registered in `TEMPLATES['DIRS']` and seeded with a minimal base template. In `headless` mode no template directory is created.

Finally, the command lists the required packages and asks whether to install them. If you agree (or run with `--noinput`), it installs them, runs the database migrations and then `cms check` to validate the installation. If you decline, it prints the commands to run later instead. Existing entries are never duplicated, so it is safe to run again.

To preview the changes without touching anything, add `--dry-run`. The command then prints a unified diff of every edit it would make to your settings and `urls.py` (and a new-file diff for any template it would create), and lists the packages it would install — but writes nothing and installs nothing:

```
djangocms . --dry-run
```

`--dry-run` only applies when adding django CMS to an existing project; it is not available when creating a new project from the template.

Unlike when creating a new project, no superuser is created (the `--username` and `--email` options have no effect); use your project's existing accounts or run `python -m manage createsuperuser`.

The exact apps, middleware, context processors, settings and url patterns to add — and the conditions under which each applies (based on `--mode` and the feature flags) — are described by a JSON rules file. Like the project template, it is fetched from the cms-template repository (the branch matching your installed django CMS version); a copy bundled with django CMS is used as an offline fallback.

> **Note**
>
> The `--stories`, `--mode`, `--versioning`, `--moderation` and `--alias` options are evaluated by the project template. They require version 5.1 or later of the cms-template to support them.

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
```

Maintenance and repair

`fix-tree`

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

This command will fix small corruptions by rebuilding the tree.

Added in version 4.0: Since django CMS Version 4 this command does not affect the plugin tree.

Configuring django CMS

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` setting

`cms.middleware.utils.ApphookReloadMiddleware`

Adding `ApphookReloadMiddleware` to the `MIDDLEWARE` 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.

Basic Customisation

CMS_TEMPLATES

default
: `()` (Valid setting for headless mode only!)

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 basic_template.

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.

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:::

```
from django.utils.translation import gettext_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_PLACEHOLDERS

default

```
(('', ('content',), _("Single placeholder")),)
```

A list of placeholders that can be added to a page. The first element of the tuple is the name of the placeholder configuration. The second element is a tuple of placeholder names. The third element is the verbose description of the placeholder configuration which will be shown in the user interface.

The `CMS_PLACEHOLDERS` setting is used to define the placeholders in headless mode if and only if no CMS templates are defined in *CMS_TEMPLATES* or *CMS_TEMPLATES_DIR*.

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'],
```

(continues on next page)

(continued from previous page)

```
        "extra_context": {"width": 280},
        'name': gettext("Right Column"),
        'limits': {
            'global': 2,
            'TeaserPlugin': 1,
            'LinkPlugin': 1,
        },
        'plugin_modules': {
            'LinkPlugin': 'Extra',
        },
        'plugin_labels': {
            'LinkPlugin': 'Add a link',
        },
    },
    'base.html content': {
        "plugins": ['TextPlugin', 'PicturePlugin', 'TeaserPlugin'],
        'inherit': 'content',
    },
}
```

You can combine template names and placeholder names to define plugins in a granular fashion, as shown above with `base.html content`.

Configuration is retrieved in the following order:

1. CMS_PLACEHOLDER_CONF['template placeholder']
2. CMS_PLACEHOLDER_CONF['placeholder']
3. CMS_PLACEHOLDER_CONF['template']
4. CMS_PLACEHOLDER_CONF[None]

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

`template` denotes a page's template setting, e.g. `pages/with_sidebar.html`. The template name must end in `.htm` or `.html`. `placeholder` is a placeholder slot name.

Changed in version 5.0: The template selector is available on django CMS pages. Since django CMS 5.0 it also is available for other models, provided they provide a `get_template()` method.

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/'
                        },
                    },
                    {
```

(continues on next page)

(continued from previous page)

```
                        '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 (or set to `None`), all plugins can be selected. An empty list (`[]`) means no plugins are allowed as children. List entries may be glob patterns such as `"Bootstrap*"` or `"*Link*"`, which are expanded against the names of all registered plugins (a pattern matching no plugin is therefore equivalent to an empty list). As a special case, the string `"auto"` allows exactly those plugins that name this plugin in their `parent_classes` – the children opt in to the parent rather than the parent listing them.

parent_classes
A dictionary of plugin names with lists describing which plugins may contain each plugin. If not supplied (or set to `None`), all plugins can be selected. An empty list (`[]`) means the plugin allows no parent and can only be added directly to a placeholder. List entries may be glob patterns such as `"Bootstrap*"` or `"*Link*"`, which are expanded against the names of all registered plugins (a pattern matching no plugin is therefore equivalent to an empty list). As a special case, `["*"]` matches every registered plugin and thus requires the plugin to have a parent (any plugin), in contrast to `None` which allows – but does not require – a parent.

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

As a rule of thumb, prefer naming specific `parent_classes` over setting `require_parent = True`. Listing concrete parents both restricts where the plugin may be added *and* requires a parent, so `require_parent` becomes redundant. If genuinely any plugin is an acceptable parent, make that explicit with `parent_classes = ["*"]` rather than relying on `require_parent` alone.

Note

For model-level and plugin-level filtering of available plugins, see *Restricting plugins to specific models* in the custom plugins documentation. The `allowed_models` attribute (on plugins) and `allowed_plugins` attribute (on models) provide fine-grained control over plugin availability beyond the placeholder-level configuration described here.

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 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/10-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'],
```

(continues on next page)

(continued from previous page)

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

Note

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

Warning

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

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

On the first level you can set values for each `SITE_ID`. In the example above we define two sites. The first site has 3 languages (English, German and French) and the second site has only Dutch.

The `default` node defines default 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.

See also *Multi-Site Installation* for patterns to run multiple sites and how per-site language configurations are selected

at runtime without a global `SITE_ID`.

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.

CMS_DEFAULT_IN_NAVIGATION

default
`True`

Decides if a newly added page content is automatically added to the navigation.

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_MENU_CACHE_BACKEND

Added in version 5.1.

default
`'default'`

The Django cache alias used for the menu tree cache. Set this to route menu cache entries to a dedicated backend.

Example:

```
CMS_MENU_CACHE_BACKEND = 'menu'

CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.PyMemcacheCache',
        'LOCATION': '127.0.0.1:11211',
    },
    'menu': {
        'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',
        'LOCATION': '/var/tmp/django_menu_cache',
    },
}
```

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',
    'cms.cms_toolbars.PageToolbar',

    # third-party Toolbar
    'aldryn_blog.cms_toolbars.BlogToolbar',
]
```

CMS_ENABLE_HELP

default
: `True`

This setting controls if the help menu appears in the toolbar.

CMS_EXTRA_HELP_MENU_ITEMS

Example:

```
CMS_EXTRA_HELP_MENU_ITEMS = (
    (_('Community forum'), 'https://discourse.django-cms.org/'),
    (_('Documentation'), 'https://docs.django-cms.org/en/latest/'),
    (_('Getting started'), 'https://www.django-cms.org/en/get-started-django-cms/'),
    (_('Talk to us'), 'https://www.django-cms.org/en/support/'),
)
```

This setting adds to the default links of the support menu allowing project or company support links.

CMS_TOOLBAR_ANONYMOUS_ON

default
: `True`

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

CMS_TOOLBAR_URL__ENABLE

default

`"toolbar_on`

This setting controls how users can activate the CMS toolbar by appending a query string to the url. The default setting lets `?toolbar_on` activate the toolbar.

> **Note**
>
> This replaces the `?edit` query string of django CMS 3.x

CMS_TOOLBAR_URL__DISABLE

default

`"toolbar_off"`

This setting controls how users can deactivate the CMS toolbar by appending a query string to the url. The default setting lets `?toolbar_off` deactivate the toolbar.

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.

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_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* ` 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* ` in your project **and** your custom plugin defined in *CMS_PAGE_WIZARD_CONTENT_PLUGIN* have a body field **different** than `body`.

CMS_ENDPOINT_LIVE_URL_QUERYSTRING_PARAM_ENABLED

default
`False`

Added in version 4.0: Setting to enable the appending of a PageContents live url to its preview and edit endpoints as a querystring parameter. This is disabled by default.

CMS_ENDPOINT_LIVE_URL_QUERYSTRING_PARAM

default
`live-url`

Added in version 4.0: Setting to configure the query string parameter name used for the live-url of a PageContent edit/preview endpoint.

CMS_REDIRECT_PRESERVE_QUERY_PARAMS

default
`False`

This indicates to the CMS that redirects should preserve the query parameters.

CMS_REDIRECT_TO_LOWERCASE_SLUG

default
`False`

This indicates to the CMS that it should redirect requests with an non-lowercase slug to its lowercase version if no page with that slug is found.

CMS_CATCH_PLUGIN_500_EXCEPTION

default
`True`

Should rendering plugins cause an exception, they are caught by default. In edit mode the exception is shown in the placeholder, in preview mode and on public content the placeholder remains empty.

If `CMS_CATCH_PLUGIN_500_EXCEPTION` is set to `False`, viewing public content will cause a server error (http error code 500). This can, for example, be used for regular health checking.

CMS_ALWAYS_REFRESH_CONTENT

default
`False`

Added in version 5.0.

If set to `True`, the CMS will always refresh the content of the page after edit action, just as in django CMS 4.1 and before.

Only use this setting if your custom plugins have issues with the new partial content refresh when editing. **If you need to set this, make sure to report an issue on GitHub.**

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.

Note

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.**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=None*, *site=None*, *login_required=False*, *limit_visibility_in_menu=None*, *position='last-child'*, *overwrite_url=None*, *xframe_options=0*, *page_title=None*)

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

Warning

Since version 4 the parameters published, publication_date, and publication_end_date do not change the behaviour of this function. If they are supplied a warning is raised.

Parameters

- **title** (*str*) – Title of the page
- **template** (*str*) – Template to use for this page. Must be in *CMS_TEMPLATES*
- **language** (*str*) – Language code for this page. Must be in LANGUAGES
- **menu_title** (*str*) – Menu title for this page
- **slug** (*str*) – Slug for the page, by default uses a slugified version of *title*
- **apphook** (str or *cms.app_base.CMSApp* sub-class) – Application to hook on this page, must be a valid apphook
- **apphook_namespace** (*str*) – Name of the apphook namespace
- **redirect** (*str*) – URL redirect
- **meta_description** (*str*) – Description of this page for SEO
- **created_by** (str of `django.contrib.auth.models.User` instance) – User that is creating this page
- **parent** (`cms.models.Page` instance) – Parent page of 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** (*str*) – Reverse ID of this page (for template tags)
- **navigation_extenders** (*str*) – Menu to attach to this page. Must be a valid menu
- **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_visibility_in_menu** (`VISIBILITY_ALL` or `VISIBILITY_USERS` or `VISIBILITY_ANONYMOUS`) – Limits visibility of this page in the menu
- **position** (*str*) – Where to insert this node if *parent* is given, must be `'first-child'` or `'last-child'`
- **overwrite_url** (*str*) – Overwritten path for this page

- **xframe_options** (*int*) – X Frame Option value for Clickjacking protection
- **page_title** (*str*) – Overridden page title for HTML title tag

cms.api.**create_page_content**(*language*, *title*, *page*, *menu_title=None*, *slug=None*, *redirect=None*, *meta_description=None*, *parent=None*, *overwrite_url=None*, *page_title=None*, *path=None*, *created_by='python-api'*, *soft_root=False*, *in_navigation=False*, *template='INHERIT'*, *limit_visibility_in_menu=None*, *xframe_options=0*)

Creates a `cms.models.PageContent` instance and returns it.

`parent` is only used if slug=None.

Parameters

- **language** (*str*) – Language code for this page. Must be in `LANGUAGES`
- **title** (*str*) – Title of the page
- **page** (`cms.models.Page` instance) – The page for which to create this title
- **menu_title** (*str*) – Menu title for this page
- **slug** (*str*) – Slug for the page, by default uses a slugified version of *title*
- **redirect** (*str*) – URL redirect
- **meta_description** (*str*) – Description of this page for SEO
- **parent** (`cms.models.Page` instance) – Used for automated slug generation
- **overwrite_url** (*str*) – Overwritten path for this page
- **page_title** (*str*) – Overridden page title for HTML title tag

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** (str or *cms.plugin_base.CMSPluginBase* sub-class, must be a valid plugin) – What type of plugin to add
- **language** (*str*) – Language code for this plugin, must be in `LANGUAGES`
- **position** (*str*) – Position to add this plugin to the placeholder. Allowed positions are `"last-child"` (default), `"first-child"`, `"left"`, `"right"`.
- **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_view_page=True*, *can_change_page=True*, *can_delete_page=True*, *can_publish_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=5*, *can_add=False*, *can_change=False*, *can_delete=False*, *can_change_advanced_settings=False*, *can_publish=None*, *can_change_permissions=False*, *can_move_page=False*, *can_recover_page=True*, *can_view=False*, *grant_all=False*, *global_permission=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,) – Controls which pages are affected

cms.models.ACCESS_DESCENDANTS or cms.models.ACCESS_PAGE_AND_DESCENDANTS :param can_*: Permissions to grant :param bool grant_all: Grant all permissions to the user

cms.api.**publish_page**(*page*, *user*, *language*)

> **Warning**
>
> Publishing pages has been removed from django CMS core in version 4 onward.
>
> For publishing functionality see djangocms-versioning:

cms.api.**publish_pages**(*include_unpublished=False*, *language=None*, *site=None*)

> **Warning**
>
> Publishing pages has been removed from django CMS core in version 4 onward.
>
> For publishing functionality see djangocms-versioning:

cms.api.**get_page_draft**(*page*)

> **Warning**
>
> The concept of draft pages has been removed from django CMS core in version 4 onward.
>
> For draft functionality see djangocms-versioning:

cms.api.**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.

Parameters

- **page** (*cms.models.pagemodel.Page* instance) – the page to copy
- **source_language** (*string*) – The source language code, must be in LANGUAGES
- **target_language** (*string*) – The source language code, must be in LANGUAGES
- **only_empty** (*bool*) – if False, plugin are copied even if plugins exists in the target language (on a placeholder basis).

Return int
number of copied plugins

cms.api.**can_change_page**(*request*)

Check whether a user has the permission to change the page.

This will work across all permission-related setting, with a unified interface to permission checking.

Parameters
request (HttpRequest instance) – The request object from which the user will be taken.

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', created_by=user)  # Since django␣
↪CMS 4, we need a user object
page_content = page.get_admin_content('en')
placeholder = page_content.get_placeholders().get(slot='body')
add_plugin(placeholder, 'TextPlugin', 'en', body='hello world')
```

cms.constants

cms.constants.**VISIBILITY_ALL = None**

Used for the `limit_visibility_in_menu` keyword argument to :func: *create_page*.Does not limit menu visibility.

cms.constants.**VISIBILITY_USERS = 1**

Used for the `limit_visibility_in_menu` keyword argument to :func: *create_page*. Limits menu visibility to authenticated users.

cms.constants.**VISIBILITY_ANONYMOUS = 2**

Used for the `limit_visibility_in_menu` keyword argument to :func: *create_page*. Limits menu visibility to anonymous(not authenticated) users.

cms.constants.**TEMPLATE_INHERITANCE_MAGIC = 'INHERIT'**

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: On the left side.

cms.constants.**RIGHT**

Used as a position indicator in the toolbar: On the right side.

cms.constants.**EXPIRE_NOW = 0**

Used for cache control headers: 0 seconds, i.e. now.

cms.constants.**MAX_EXPIRATION_TTL = 31536000**

Used for cache control headers: 365 * 24 * 3600 seconds, i.e. one year. HTTP specification says max caching should only be up to one year.

Configuring apps to work with django CMS

App Hooks

class cms.app_base.**CMSApp**

Base class for creating apphooks. Apphooks live in a file called `cms_apps.py`. To create an AppHook subclass `CMSApp` in `cms_apps.py`

```
class MyAppHook(CMSApp):
    name = "Problem solver"
```

_urls

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

_menus

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

get_config(*namespace*)

Returns the apphook configuration instance linked to the given namespace

To be implemented by apphook subclass.

get_config_add_url()

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

To be implemented by apphook subclass.

get_configs()

Returns all the apphook configuration instances.

To be implemented by apphook subclass.

get_menus(*page=None*, *language=None*, ***kwargs*)

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

By default, it returns the menus assigned to *CMSApp._menus*.

The method accepts page, language and generic keyword arguments: you can customize this function to return different list of menu classes according to the given arguments.

If no menus are returned, then the user will need to attach menus to pages manually in the admin.

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]
```

Parameters

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

Returns
list of menu classes

`get_root_template`(*page=None*, *language=None*, ***kwargs*)

Added in version 5.1.

Returns the template name used by the apphook's root view.

Best-effort: walks the urlconfs from *get_urls()* to locate the pattern matching the empty path and returns `template_name` from its view class (CBVs) or callback (FBVs that expose it). Returns `None` for views that compute the template dynamically — override this method to return the template name in that case.

Returns
template name or None

`get_urls`(*page=None*, *language=None*, ***kwargs*)

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

By default, it returns the urls assigned to *CMSApp._urls*

The method accepts page, language and generic keyword arguments: you can customize this function to return different urlconfs according to the given arguments.

This method **must** return a non-empty list of urlconfs, even if no argument is passed.

Parameters

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

Returns
list of urlconfs strings

`app_config = None`

configuration model (optional)

`app_name = None`

Gives the system a unique way to refer to the apphook. This enables Django namespaces support (optional)

`exclude_permissions = []`

list of application names to exclude from inheriting CMS permissions

`name = None`

Human-readable name of the apphook (required). This name will be displayed on the admin site.

`permissions = True`

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

App Config

`class` `cms.app_base.`**`CMSAppConfig`**(*django_app_config*)

Added in version 4.0.

Base class that all cms app configurations should inherit from.

CMSAppConfig live in a file called `cms_config.py`.

Apps subclassing `CMSAppConfig` can set `cms_enabled = True` for their app config to use django CMS' wizard functionality. Additional wizzwards are listed in the app config's `cms_wizzards` property.

The second functionality that django CMS offers is attaching Model objects to the toolbar. To use this functionality, set list the Model classes in `cms_toolbar_enabled_models` and have `cms_enabled = True`

static get_contract(*contract_name: str*) → type

Retrieve a contract implementation from registered CMS extension apps.

Searches through all registered CMS extension applications for a contract that matches the given contract name. A contract is a named interface that allows apps to provide functionality that other apps can use.

This method enables an architecture where apps can query for specific contracts and use the implementation provided by any registered extension app that exports that contract.

Parameters
contract_name (*str*) – The name identifier of the contract to retrieve

Returns
The contract class if found, or None if no matching contract exists

Return type
type or None

Raises
None

Example:

```
# Get a versioning contract implementation
versioning_contract = CMSAppConfig.get_contract('versioning')
if versioning_contract:
    # Use the versioning functionality
    version = versioning_contract.create_version(obj)
```

App Extensions

class cms.app_base.**CMSAppExtension**

Added in version 4.0.

Base class that all cms app extensions should inherit from. App extensions allow apps to offer their functionality to other apps, e.g., as done by djangocms-versioning.

CMSAppExtensions live in a file called `cms_config.py`.

abstractmethod configure_app(*cms_config*)

Implement this method if the app provides functionality that other apps can use and configure.

This method will be run once for every app that defines an attribute like `<app_label>_enabled` as `True` on its cms app config class.

So for example, if app A with label "app_a" implements this method and app B and app C define `app_a_enabled = True` on their cms config classes, the method app A has defined will run twice, once for app B and once for app C.

Parameters
cms_config (*CMSAppConfig* subclass) – the cms config class of the app registering for additional functionality

ready()

Override this method to run code after all CMS extensions have been configured.

This method will be run once, even if no cms app config sets its <app_label>_enabled attribute to True

Form and model fields

Model fields

class cms.models.fields.**PageField**(***kwargs*)

Bases: ForeignKey

This is a foreign key field to the *cms.models.pagemodel.Page* model that defaults to the *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.

default_form_class

alias of *PageSelectFormField*

formfield(***kwargs*)

Pass limit_choices_to to the field being constructed.

Only passes it if there is a type that supports related fields. This is a similar strategy used to pass the queryset to the field being constructed.

class cms.models.fields.**PlaceholderRelationField**(*checks=None*, ***kwargs*)

Bases: GenericRelation

GenericForeignKey to placeholders.

If you create a model which contains placeholders you first create the PlaceHolderRelationField:

```
from cms.utils.placeholder import get_placeholder_from_slot

class Post(models.Model):
    ...
    placeholders = PlaceholderRelationField()  # Generic relation

    @cached_property
    def content(self):
        return get_placeholder_from_slot(self.placeholders, "content")  # A␣
↪specific placeholder
```

class cms.models.fields.**PlaceholderField**(*slotname*, **args*, ***kwargs*)

> **Warning**
>
> This field is for django CMS versions below 4 only. It may only be used inside migrations.

The PlaceholderField has been replaced by the *PlaceholderRelationField*, the built-in migrations will automatically take care of the replacement. See documentation of *PlaceholderRelationField* for how to replace the code.

Form fields

class cms.forms.fields.**PageSelectFormField**(*queryset=None*, *empty_label='---------'*, *cache_choices=False*, *required=True*, *widget=None*, *to_field_name=None*, *limit_choices_to=None*, **args*, ***kwargs*)

Behaves like a `django.forms.ModelChoiceField` field for the *cms.models.pagemodel.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`.

widget

alias of `PageSelectWidget`

compress(*data_list*)

Return a single value for the given list of values. The values can be assumed to be valid.

For example, if this MultiValueField was instantiated with fields=(DateField(), TimeField()), this might return a datetime object created by combining the date and time in data_list.

has_changed(*initial*, *data*)

Return True if data differs from initial.

class cms.forms.fields.**PageSmartLinkField**(*max_length=None*, *min_length=None*, *placeholder_text=None*, *ajax_view=None*, **args*, ***kwargs*)

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.

> **Note**
>
> `PageSmartLinkField` stores its value as a plain URL string, so links do not follow pages that are later moved or renamed. For dynamically linking to pages, other CMS content, or external URLs, the preferred approach is the `LinkField` provided by djangocms-link, which keeps references stable across moves and supports a configurable set of link targets.

widget

alias of `PageSmartLinkWidget`

clean(*value*)

Validate the given value and return its "cleaned" value as an appropriate Python object. Raise ValidationError for any errors.

widget_attrs(*widget*)

Given a Widget instance (*not* a Widget class), return a dictionary of any HTML attributes that should be added to the Widget, based on this Field.

User site 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'.

Menu system classes and function

class menus.base.**Menu**(*renderer*)

The base class for all menu-generating classes.

get_nodes(*request*) → list[*NavigationNode*]

Get a list of NavigationNode instances for the menu.

Args:
request: The request object.

Returns:
A list of NavigationNode instances.

class menus.base.**Modifier**(*renderer*)

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

modify(*request*, *nodes*, *namespace*, *root_id*, *post_cut*, *breadcrumb*)

Modify the list of nodes.

Args:
request: The request object. nodes: List of NavigationNode instances. namespace: The namespace for the menu. root_id: ID of the root node. post_cut: Boolean indicating post-cut status. breadcrumb: Boolean indicating breadcrumb status.

class menus.base.**NavigationNode**(*title: str*, *url: str*, *id: Any*, *parent_id: Any | None = None*, *parent_namespace: str | None = None*, *attr: dict[str, Any] | None = None*, *visible: bool = True*)

Represents each node in a menu tree.

Attributes:
title: The title of the menu item. url: The URL associated with the menu item. id: The unique ID of this

item. parent_id: The ID of the parent item (optional). parent_namespace: The namespace of the parent (optional). attr: Additional information to store on this node (optional). visible: Indicates whether this item is visible (default is True).

get_absolute_url() → str

Returns the URL associated with this menu item.

get_ancestors() → list[*NavigationNode*]

Returns a list of all parent items, excluding the current menu item.

get_attribute(*name: str*) → Any

Retrieves a dictionary item from 'attr'. Returns None if it does not exist.

Args:
name: The name of the attribute.

Returns:
The value associated with the attribute name or None if not found.

get_descendants() → list[*NavigationNode*]

Returns a list of all children beneath the current menu item.

get_menu_title() → str

Returns the associated title using the naming convention of 'cms.models.pagemodel.Page'.

is_selected(*request*) → bool

Checks if the node is selected based on the request path.

Args:
request: The request object.

Returns:
True if the node is selected, False otherwise.

attr

A dictionary to add arbitrary attributes to the node. An important key is 'is_page': * If True, the node represents a django CMS 'Page' object. * Nodes representing CMS pages have specific keys in 'attr'.

property is_leaf_node: bool

Indicates whether the node is a leaf node.

class menus.menu_pool.**MenuPool**

clear(*site_id=None*, *language=None*, *all=False*)

This invalidates the cache for a given menu (site_id and language)

get_menus_by_attribute(*name*, *value*)

Returns the list of menus that match the name/value criteria provided.

get_registered_menus(*for_rendering=False*)

Returns all registered menu classes.

Parameters
for_rendering – Flag that when True forces us to include all CMSAttachMenu subclasses, even if they're not attached.

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: str*, *url: str*, *id: Any*, *parent_id: Any | None = None*, *parent_namespace: str | None = None*, *attr: dict[str, Any] | None = None*, *visible: bool = True*)

Represents each node in a menu tree.

Attributes:

title: The title of the menu item. url: The URL associated with the menu item. id: The unique ID of this item. parent_id: The ID of the parent item (optional). parent_namespace: The namespace of the parent (optional). attr: Additional information to store on this node (optional). visible: Indicates whether this item is visible (default is True).

__init__(*title: str*, *url: str*, *id: Any*, *parent_id: Any | None = None*, *parent_namespace: str | None = None*, *attr: dict[str, Any] | None = None*, *visible: bool = True*)

Initialize a NavigationNode instance.

Args:

title: The title of the menu item. url: The URL associated with the menu item. id: The unique ID of this item. parent_id: The ID of the parent item (optional). parent_namespace: The namespace of the parent (optional). attr: Additional information to store on this node (optional). visible: Indicates whether this item is visible (default is True).

get_absolute_url() → str

Returns the URL associated with this menu item.

get_ancestors() → list[*NavigationNode*]

Returns a list of all parent items, excluding the current menu item.

get_attribute(*name: str*) → Any

Retrieves a dictionary item from 'attr'. Returns None if it does not exist.

Args:

name: The name of the attribute.

Returns:

The value associated with the attribute name or None if not found.

get_descendants() → list[*NavigationNode*]

Returns a list of all children beneath the current menu item.

get_menu_title() → str

Returns the associated title using the naming convention of 'cms.models.pagemodel.Page'.

is_selected(*request*) → bool

Checks if the node is selected based on the request path.

Args:
request: The request object.

Returns:
True if the node is selected, False otherwise.

attr

A dictionary to add arbitrary attributes to the node. An important key is 'is_page': * If True, the node represents a django CMS 'Page' object. * Nodes representing CMS pages have specific keys in 'attr'.

property is_leaf_node: bool

Indicates whether the node is a leaf node.

class menus.modifiers.**AuthVisibility**(*renderer*)

Remove nodes that are login required or require a group

modify(*request*, *nodes*, *namespace*, *root_id*, *post_cut*, *breadcrumb*)

Modify the list of nodes based on certain conditions.

Parameters

- **self** – The instance of the class containing this method.
- **request** – The current request object.
- **nodes** (*list*) – A list of nodes to be modified.
- **namespace** – The namespace.
- **root_id** – The ID of the root node.
- **post_cut** (*bool*) – Flag indicating if the modification is happening after cutting.
- **breadcrumb** (*bool*) – Flag indicating if the modification is happening for the breadcrumb.

Returns
The modified list of nodes.

Return type
list

class menus.modifiers.**Level**(*renderer*)

Marks all node levels.

mark_levels(*node*, *post_cut*)

Mark the levels of menu items.

Parameters

- **node** (*Node*) – The root node of the menu hierarchy.
- **post_cut** (*bool*) – Flag indicating whether the function is called after a cut is made.

Returns
None

Return type
None

Raises
None – This function does not raise any exceptions.

modify(*request*, *nodes*, *namespace*, *root_id*, *post_cut*, *breadcrumb*)

Modify the list of nodes based on certain conditions.

Parameters

- **self** – The instance of the class containing this method.
- **request** – The current request object.
- **nodes** (*list*) – A list of nodes to be modified.
- **namespace** – The namespace.
- **root_id** – The ID of the root node.
- **post_cut** (*bool*) – Flag indicating if the modification is happening after cutting.
- **breadcrumb** (*bool*) – Flag indicating if the modification is happening for the breadcrumb.

Returns

The modified list of nodes.

Return type

list

CMS menus

cms application

class cms.cms_menus.**CMSMenu**(*renderer*)

Subclass of *menus.base.Menu*. Its *get_nodes()* creates a list of NavigationNodes based on a site's *cms.models.pagemodel.Page* objects.

get_menu_node_for_page_content(*page_content: PageContent*, *view_url: str | None = None*, *cut: bool = False*) → CMSNavigationNode

Transform a CMS page content object into a navigation node.

Parameters

- **page** – The page to transform.
- **languages** – The list of the current language plus fallbacks used to render the menu.
- **view_url** – If given, serves as a "pattern" for a view url with the assumption that "/0/" is replaced by the actual page content pk. Default is None.
- **cut** – If True the parent_id is set to None. Default is False.

Returns

A CMSNavigationNode instance.

get_nodes(*request*) → list[*NavigationNode*]

Returns a list of NavigationNode objects representing the navigation nodes to be displayed in the menu. This method is performance-critical since the number of page content objects can be large.

Parameters

- **self** – The instance of the class.
- **request** – The HTTP request object.

Returns

A list of NavigationNode objects representing the navigation nodes.

Return type
list[*NavigationNode*]

Note

- The method retrieves the necessary data from the database to build the navigation nodes for the menu.
- The behavior of the method depends on whether the edit mode or preview mode is active in the toolbar.
- If either edit mode or preview mode is active, the method retrieves all current page content objects visible in the admin for the current page.
- If neither edit mode nor preview mode is active, the method retrieves only public page content objects.
- The retrieved page contents are filtered based on the specified languages, sorted by page path, and filtered by site.
- Only specific fields of the page content objects are selected to optimize performance.
- If either edit mode or preview mode is active, a preview URL is constructed for a "virtual" non-existing page content with id=0 to avoid too many calls to `revert` the admin URL.
- The method includes a nested function for prefetching URLs and filling the URL cache.
- The visibility of the page contents is further filtered based on authentication and permissions.
- The homepage is determined based on the page contents and marked for cutting if necessary.
- The menu node for each page content is created using the get_menu_node_for_page_content method of the instance.
- The prefetch_urls function is called for each page content to fill the URL cache and provide necessary data for creating the menu node.
- The select_lang method is used to filter the page contents based on the specified language preferences.

select_lang(*page_contents: Iterable[*PageContent*]*) → Generator[*PageContent*, None, None]

Generator that returns only those page content objects passed that contain the first language present in the languages list.

class cms.cms_menus.**NavExtender**(*renderer*)

modify(*request*, *nodes*, *namespace*, *root_id*, *post_cut*, *breadcrumb*)

Modify the list of nodes.

Args:
request: The request object. nodes: List of NavigationNode instances. namespace: The namespace for the menu. root_id: ID of the root node. post_cut: Boolean indicating post-cut status. breadcrumb: Boolean indicating breadcrumb status.

class cms.cms_menus.**SoftRootCutter**(*renderer*)

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

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

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

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

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

- **School of Medicine**
  - Medical Education
  - **Departments**
    - Department of Lorem Ipsum
    - Department of Donec Imperdiet
    - Department of Cras Eros
    - **Department of Mediaeval Surgery**
      - Theory
      - Cures
      - **Bleeding**

        Introduction to Bleeding <this is the current page>

        Bleeding - the scientific evidence

        Cleaning up the mess

        Cupping

        Leaches

        Maggots
      - Techniques
      - Instruments
    - Department of Curabitur a Purus
    - Department of Sed Accumsan
    - Department of Etiam
  - Research
  - Administration
  - Contact us
  - Impressum

which is frankly overwhelming.

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

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

        * Cupping
        * Leaches
        * Maggots
    - Techniques
    - Instruments

find_ancestors_and_remove_children(*node*, *nodes*)

Check ancestors of node for soft roots

modify(*request*, *nodes*, *namespace*, *root_id*, *post_cut*, *breadcrumb*)

Modify the list of nodes.

Args:

request: The request object. nodes: List of NavigationNode instances. namespace: The namespace for the menu. root_id: ID of the root node. post_cut: Boolean indicating post-cut status. breadcrumb: Boolean indicating breadcrumb status.

class cms.menu_bases.**CMSAttachMenu**(**args*, ***kwargs*)

Base class that can be subclassed to allow your app to attach its own menus.

classmethod get_apphooks()

Returns a list of apphooks to which this CMSAttachMenu is attached.

Calling this does **not** produce DB queries.

classmethod get_instances()

Return a queryset of all CMS Page objects (in this case) that are currently using this CMSAttachMenu either directly as a navigation_extender, or, as part of an apphook.

Calling this **does** perform a DB query.

Pages

class cms.models.pagemodel.**Page**(**args*, ***kwargs*)

Bases: MP_Node

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 is an abstract entity. It does not have any content associated with it, nor does is provide any slugs to build a URL. This is part of the model *PageContent* and *PageUrl* respectively.

add_child(***kwargs*)

Adds a child to the node.

This method saves the node in database. The object is populated as if via:

` obj = self.__class__(**kwargs) `

Raises

PathOverflow – when no more child nodes can be added

add_sibling(*pos=None*, **args*, ***kwargs*)

Adds a new node as a sibling to the current node object.

This method saves the node in database. The object is populated as if via:

` obj = self.__class__(**kwargs) `

Raises

PathOverflow – when the library can't make room for the node's new position

copy_with_descendants(*target_page=None*, *target_node=None*, *position=None*, *copy_permissions=True*, *target_site=None*, *user=None*)

Copy a page [and all its descendants to a new location]

delete(**args*, ***kwargs*)

Removes a node and all it's descendants.

get_application_urls(*language=None*, *fallback=True*, *force_reload=False*)

get application urls conf for application hook

get_changed_by(*language=None*, *fallback=True*, *force_reload=False*)

get user who last changed this page

get_changed_date(*language=None*, *fallback=True*, *force_reload=False*)

get when this page was last updated

get_content_obj(*language=None*, *fallback=True*, *force_reload=False*)

Helper function for accessing wanted / current title. If wanted title doesn't exist, EmptyPageContent instance will be returned.

get_languages(*admin_manager=True*)

Returns available languages for the page. This is potentially costly.

get_media_path(*filename*)

Returns path (relative to MEDIA_ROOT/MEDIA_URL) to directory for storing page-scope files. This allows multiple pages to contain files with identical names without namespace issues. Plugins such as Picture can use this method to initialise the 'upload_to' parameter for File-based fields. For example:

image = models.ImageField(
_("image"), upload_to=CMSPlugin.get_media_path)

where CMSPlugin.get_media_path calls self.page.get_media_path

This location can be customised using the CMS_PAGE_MEDIA_PATH setting

get_menu_title(*language=None*, *fallback=True*, *force_reload=False*)

get the menu title of the page depending on the given language

get_meta_description(*language=None*, *fallback=True*, *force_reload=False*)

get content for the description meta tag for the page depending on the given language

get_page_content_obj_attribute(*attrname*, *language=None*, *fallback=True*, *force_reload=False*)

Helper function for getting attribute or None from wanted/current page content.

get_page_title(*language=None*, *fallback=True*, *force_reload=False*)

get the page title of the page depending on the given language

get_redirect(*language=None*, *fallback=True*, *force_reload=False*)

get redirect

get_root()

Returns
the root node for the current node object.

get_template_name()

get the textual name (2nd parameter in get_cms_setting('TEMPLATES')) of the template of this page or of the nearest ancestor. failing to find that, return the name of the default template.

get_title(*language=None*, *fallback=True*, *force_reload=False*)

get the title of the page depending on the given language

get_url_obj(*language*, *fallback=True*)

Get the path of the page depending on the given language

has_add_permission(*user*)

Has user ability to add page under current page?

has_change_permissions_permission(*user*)

Has user ability to change permissions for current page?

has_move_page_permission(*user*)

Has user ability to move current page?

is_potential_home()

Encapsulates logic for determining if this page is eligible to be set as *is_home*. This is a public method so that it can be accessed in the admin for determining whether to enable the "Set as home" menu item. :return: Boolean

move_page(*target_page*, *position='first-child'*)

Called from admin interface when page is moved. Should be used on all the places which are changing page position. Used like an interface to django-treebeard.

reload()

Reload a page from the database

save(***kwargs*)

Save the current instance. Override this in a subclass if you want to control the saving process.

The 'force_insert' and 'force_update' parameters can be used to insist that the "save" must be an SQL insert or update (or equivalent for non-SQL backends), respectively. Normally, they should not be set.

set_as_homepage(*user=None*)

Sets the given page as the homepage. Updates the url paths for all affected pages. Returns the old home page (if any).

admin_content_cache

Internal cache for page content objects visible publicly

page_content_cache

Internal cache for page urls

class cms.models.pagemodel.**PageUrl**(*id*, *slug*, *path*, *language*, *page*, *managed*)

Bases: Model

class cms.models.pagemodel.**PageType**(*id*, *path*, *depth*, *numchild*, *parent*, *site*, *created_by*, *changed_by*, *creation_date*, *changed_date*, *reverse_id*, *navigation_extenders*, *login_required*, *is_home*, *application_urls*, *application_namespace*, *is_page_type*)

Bases: *Page*

is_potential_home()

Encapsulates logic for determining if this page is eligible to be set as *is_home*. This is a public method so that it can be accessed in the admin for determining whether to enable the "Set as home" menu item. :return: Boolean

.

Page contents

class cms.models.contentmodels.**PageContent**(*id*, *language*, *title*, *page_title*, *menu_title*, *meta_description*, *redirect*, *page*, *creation_date*, *created_by*, *changed_by*, *changed_date*, *in_navigation*, *soft_root*, *template*, *limit_visibility_in_menu*, *xframe_options*)

content_indicator()

returns the content indicator status. Without additional packages like djangocms-versioning page content always is public.

Return type
str

get_absolute_url(*language=None*)

Get the absolute url for the page content. If language is specified it will return the absolute url of the corresponding "sister" content.

get_placeholder_slots()

Returns a list of placeholder slots for this page content object.

get_template(*app_hooks=True*)

get the template of this page if defined or if closer parent if defined or DEFAULT_PAGE_TEMPLATE otherwise.

If the page has an apphook assigned and its root view exposes a template, that template wins — it's what the apphook actually renders at the page URL, so the structure board's placeholder layout must match it.

get_template_name()

get the textual name (2nd parameter in get_cms_setting('TEMPLATES')) of the template of this title. failing to find that, return the name of the default template.

get_xframe_options()

Finds X_FRAME_OPTION from tree if inherited

is_editable(*request*)

returns True if page content object itself can be edited. Does not check user permissions to do that.

Return type
bool

rescan_placeholders()

Rescan and if necessary create placeholders in the current template.

save(***kwargs*)

Save the current instance. Override this in a subclass if you want to control the saving process.

The 'force_insert' and 'force_update' parameters can be used to insist that the "save" must be an SQL insert or update (or equivalent for non-SQL backends), respectively. Normally, they should not be set.

toggle_in_navigation(*set_to=None*)

Toggles (or sets) in_navigation and invalidates the cms page cache

admin_manager = <cms.models.managers.ContentAdminManager object>

Admin_manager does lack additional functionality of objects and must only be used inside admin objects or admin forms. One of its key properties is that it can access all objects of type PageContent (irrespevtively of some objects being hidden by third-party packages, e.g. due to viewing rights, publication or moderation status.

class cms.models.contentmodels.**EmptyPageContent**(*language*, *page=None*)

Empty title object, can be returned from *cms.models.pagemodel.Page.get_content_obj()* if required title object doesn't exist.

content_indicator()

returns the content indicator status. Empty page content always is empty

Return type

str

is_editable(*request*)

returns True if empty page content object itself can be edited. Since editing creates a new page content object this should always be True

Return type

bool

Page extensions and page content extensions

Extensions

class cms.extensions.models.**PageExtension**(**args*, ***kwargs*)

class cms.extensions.models.**PageContentExtension**(**args*, ***kwargs*)

Admin

class cms.extensions.admin.**PageExtensionAdmin**(*model*, *admin_site*)

add_view(*request*, *form_url=''*, *extra_context=None*)

Check if the page already has an extension object. If so, redirect to edit view instead.

delete_model(*request*, *obj*)

Given a model instance delete it from the database.

get_model_perms(*request*)

Return empty perms dict thus hiding the model from admin index.

save_model(*request*, *obj*, *form*, *change*)

Given a model instance save it to the database.

class cms.extensions.admin.**PageContentExtensionAdmin**(*model*, *admin_site*)

add_view(*request*, *form_url=''*, *extra_context=None*)

Check if the page already has an extension object. If so, redirect to edit view instead.

delete_model(*request*, *obj*)

Given a model instance delete it from the database.

get_model_perms(*request*)

Return empty perms dict thus hiding the model from admin index.

save_model(*request*, *obj*, *form*, *change*)

Given a model instance save it to the database.

Toolbar

class cms.extensions.toolbar.**ExtensionToolbar**(*request*, *toolbar*, *is_current_app*, *app_path*)

Offers simplified API for providing the user access to the admin of page extensions and page content extensions through the toolbar.

get_page_content_extension_admin(*page_content_obj=None*)

Get the admin url for the page content extensions menu item, depending on whether a *PageContentExtension* instance exists for the *PageContent* displayed.

Return a tuple of the page content extension and the url; the extension is None if no instance exists, the url is None is no admin is registered for the extension.

get_page_extension_admin()

Get the admin url for the page extension menu item, depending on whether a PageExtension instance exists for the current page or not.

Return a tuple of the current extension and the url; the extension is None if no instance exists, the url is None is no admin is registered for the extension.

Permissions

class cms.models.permissionmodels.**AbstractPagePermission**(**args*, ***kwargs*)

Bases: Model

Abstract page permissions

clean()

Hook for doing any extra model-wide validation after clean() has been called on every field by self.clean_fields. Any ValidationError raised by this method will not be associated with a particular field; it will have a special-case association with the field defined by NON_FIELD_ERRORS.

save(**args*, ***kwargs*)

Save the current instance. Override this in a subclass if you want to control the saving process.

The 'force_insert' and 'force_update' parameters can be used to insist that the "save" must be an SQL insert or update (or equivalent for non-SQL backends), respectively. Normally, they should not be set.

property audience

Return audience by priority, so: All or User, Group

class cms.models.permissionmodels.**PagePermission**(**args*, ***kwargs*)

Bases: *AbstractPagePermission*

Page permissions for a single page

clean()

Hook for doing any extra model-wide validation after clean() has been called on every field by self.clean_fields. Any ValidationError raised by this method will not be associated with a particular field; it will have a special-case association with the field defined by NON_FIELD_ERRORS.

class cms.models.permissionmodels.**GlobalPagePermission**(**args*, ***kwargs*)

Bases: *AbstractPagePermission*

Permissions for all pages (global).

cms.models.permissionmodels.**ACCESS_PAGE = 1**

Access to the page itself

cms.models.permissionmodels.**ACCESS_CHILDREN = 2**

Access to immediate children (1 level)

cms.models.permissionmodels.**ACCESS_DESCENDANTS = 4**

Access to all children (first level and also their children)

cms.models.permissionmodels.**ACCESS_PAGE_AND_DESCENDANTS = 5**

Access to page itself and all children (first level and also their children)

Placeholders

class cms.models.placeholdermodel.**Placeholder**(**args*, ***kwargs*)

Bases: `Model`

`Placeholders` can be filled with plugins, which store or generate content.

add_plugin(*instance:* CMSPlugin)

Added in version 4.0.

Adds a plugin to the placeholder. The plugin's position field must be set to the target position. Positions are enumerated from the start of the placeholder's plugin tree (1) to the last plugin (*n*, where *n* is the number of plugins in the placeholder).

It is discouraged to call this method directly from outside the CMS. Use the *cms.api.add_plugin()* function instead.

Parameters

instance (*cms.models.pluginmodel.CMSPlugin* instance) – Plugin to add. It's position parameter needs to be set.

> **Note**
>
> As of version 4 of django CMS the position counter does not re-start at 1 for the first child plugin. The `position` field and `language` field are unique for a placeholder.

Example:

```
new_child = MyCoolPlugin()
new_child.position = (
    parent_plugin.position + 1
)  # add as first child: directly after parent
parent_plugin.placeholder.add(new_child)
```

clear(*language=None*)

Deletes all plugins from the placeholder

delete_plugin(*instance:* CMSPlugin) → None

Added in version 4.0.

Removes a plugin and its descendants from the placeholder and database. This method **must** be used to preserver plugin tree consistency. Do not use plugin.delete() or queryset.delete()

Parameters

instance (*cms.models.pluginmodel.CMSPlugin* instance) – Plugin to remove. Its position needs to be set.

delete_plugins(*instances: Iterable[CMSPlugin]*) → None

Added in version 5.1.

Removes several plugins and their descendants from the placeholder and database in a single delete, then re-compacts the positions of each affected language once. Prefer this over repeated *delete_plugin()* calls when removing a batch of plugins.

The plugins must belong to this placeholder. Overlapping selections are fine: passing a plugin together with one of its descendants simply deletes each row once.

Parameters

instances (iterable of *cms.models.pluginmodel.CMSPlugin* instances) – Iterable of plugins to remove.

get_cache_expiration(*request*, *response_timestamp*)

Returns the number of seconds (from «response_timestamp») that this placeholder can be cached. This is derived from the plugins it contains.

This method must return: `EXPIRE_NOW <= int <= MAX_EXPIRATION_IN_SECONDS`

Return type

int

get_filled_languages(*site_id=None*)

Returns language objects for every language for which the placeholder has plugins.

This is not cached as it's meant to be used in the frontend editor.

get_next_plugin_position(*language*, *parent=None*, *insert_order='first'*)

Added in version 4.0.

Helper to calculate plugin positions correctly.

Parameters

- **language** (*str*) – language for which the position is to be calculated
- **parent** (*cms.models.pluginmodel.CMSPlugin* instance) – Parent plugin or `None` (if position is on top level)
- **insert_order** (*str*) – Either `"first"` (default) or `"last"`

get_plugin_tree_order(*language*, *parent_id=None*)

Returns a list of plugin ids matching the given language ordered by plugin position.

get_plugins(*language=None*)

Returns a queryset of plugins attached to this placeholder. If language is given only plugins in the given language are returned.

get_plugins_list(*language=None*)

Returns a list of plugins attached to this placeholder. If language is given only plugins in the given language are returned.

get_vary_cache_on(*request*)

Returns a list of VARY headers.

has_add_plugin_permission(*user*, *plugin_type*)

Returns `True` if user has permission to add `plugin_type` to this placeholder.

has_add_plugins_permission(*user*, *plugins*)

Returns `True` if user has permission to add **all** plugins in `plugins` to this placeholder.

has_change_permission(*user*)

Returns `True` if user has permission to change all models attached to this placeholder.

has_change_plugin_permission(*user*, *plugin*)

Returns `True` if user has permission to change `plugin` to this placeholder.

has_clear_permission(*user*, *languages*)

Returns `True` if user has permission to delete all plugins in this placeholder

has_delete_plugin_permission(*user*, *plugin*)

Returns `True` if user has permission to delete `plugin` to this placeholder.

has_delete_plugins_permission(*user*, *languages*)

Returns `True` if user has permission to delete all plugins in this placeholder

has_move_plugin_permission(*user*, *plugin*, *target_placeholder*)

Returns `True` if user has permission to move `plugin` to the `target_placeholder`.

has_plugins(*language=None*)

Checks if placeholder is empty (`False`) or populated (`True`)

move_plugin(*plugin*, *target_position*, *target_placeholder=None*, *target_plugin=None*)

Added in version 4.0.

Moves a plugin within the placeholder (`target_placeholder=None`) or to another placeholder.

Parameters

- **plugin** (*cms.models.pluginmodel.CMSPlugin* instance) – Plugin to move
- **target_position** (*int*) – The plugin's new position
- **target_placeholder** (*cms.models.placeholdermodel.Placeholder* instance) – Placeholder to move plugin to (or `None`)
- **target_plugin** (*cms.models.pluginmodel.CMSPlugin* instance) – New parent plugin (or `None`). The target plugin must be in the same placeholder or in the `target_placeholder` if one is given.

The `target_position` is enumerated from the start of the palceholder's plugin tree (1) to the last plugin (*n*, where *n* is the number of plugins in the placeholder).

cache_placeholder = True

Flag caching the palceholder's content

default_width

A default width is passed to the templace context as `width`

edit_message = 'Edit object'

Message used in the frontend editing UI for static placeholder edit link

is_editable = True

If False the content of the placeholder is not editable in the frontend

is_static = False

Set to "True" for static placeholders (by the template tag)

property page

Gives the page object if the placeholder belongs to a `cms.models.titlemodels.PageContent` object (and not to some other model.) If the placeholder is not attached to a page it returns `None`

slot

slot name that appears in the frontend

static_message = 'This is an external placeholder'

Message used in the frontend editing UI for static placeholder tooltip

class cms.admin.placeholderadmin.**FrontendEditableAdminMixin**

Adding `FrontendEditableAdminMixin` to models admin class allows to open that admin in the frontend by double-clicking on fields rendered with the `render_model` template tag.

get_urls() → list[str]

Register the url for the edit field view

Plugins

class cms.plugin_base.**CMSPluginBase**(*model=None*, *admin_site=None*)

Inherits `django.contrib.admin.ModelAdmin` and in most respects behaves like a normal subclass.

Note however that some attributes of `ModelAdmin` simply won't make sense in the context of a Plugin.

get_render_template(*self*, *context*, *instance*, *placeholder*)

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()*

allowed_models

A list of model identifiers (in the format `"app_label.modelname"`) that restricts where this plugin can be used. If `None` (default), the plugin is available for all models with placeholders.

See *Restricting plugins to specific models* for details.

model

If the plugin requires per-instance settings, then this setting must be set to a model that inherits from *CMSPlugin*. See also: *Storing configuration*.

alias of *CMSPlugin*

classmethod get_child_class_overrides(*slot: str*, *page: Page | None = None*, *instance: CMSPlugin | None = None*)

Returns a list of plugin types that are allowed as children of this plugin.

classmethod get_child_classes(*slot*, *page: Page | None = None*, *instance: CMSPlugin | None = None*, *only_uncached: bool = False*) → list[str]

Returns a list of plugin types that can be added as children to this plugin.

classmethod get_child_plugin_candidates(*slot: str*, *page: Page | None = None*) → list

Returns a list of all plugin classes that will be considered when fetching all available child classes for this plugin.

classmethod get_empty_change_form_text(*obj=None*)

Returns the text displayed to the user when editing a plugin that requires no configuration.

classmethod get_extra_placeholder_menu_items(*request: HttpRequest*, *placeholder:* Placeholder) → list[*PluginMenuItem*] | None

Extends the placeholder 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.

classmethod get_extra_plugin_menu_items(*request: HttpRequest*, *plugin:* CMSPlugin) → list[*PluginMenuItem*] | None

Extends the plugin 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_cache_expiration(*request*, *instance*, *placeholder*)

Provides hints to the placeholder, and in turn to the page for determining the appropriate Cache-Control headers to add to the HTTPResponse object.

Parameters

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

Return type

`None` or `datetime` or `time_delta` or `int`

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.

Datetime.timedelta

A timedelta instance indicating how long, relative to the response timestamp that the content can be cached;

Int

An integer 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.

`get_fieldsets`(*request*, *obj=None*)

Same as from base class except if there are no fields, show an info message.

`get_plugin_urls`() → list

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_vary_cache_on`(*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.

> **Note**
>
> This only makes sense to use with caching. If this plugin has `cache = False` or plugin.get_cache_expiration(...) returns 0, get_vary_cache_on() will have no effect.

`icon_alt`(*instance*)

Overwrite this if necessary if `text_enabled = True` Return the 'alt' text to be used for an icon representing the plugin object in a text editor.

Parameters
: **instance** (*cms.models.pluginmodel.CMSPlugin* instance) – The instance of the plugin model to provide specific information for the 'alt' text.

By default, *icon_alt()* will return a string of the form: "[plugin type] - [instance]", but can be modified to return anything you like.

This function accepts the `instance` as a parameter and returns a string to be used as the `alt` text for the plugin's preview or icon.

Authors of text-enabled plugins should consider overriding this function as it will be rendered as a tooltip in most browser. This is useful, because if the same plugin is used multiple times, this tooltip can provide information about its configuration.

See also: *text_enabled*, *icon_src()*.

`icon_src`(*instance*)

Deprecated: Since djangocms-text-ckeditor introduced inline previews of plugins, the icon will not be rendered in TextPlugins anymore.

By default, this returns an empty string, which, if left un-overridden would result in no icon rendered at all, which, in turn, would render the plugin un-editable 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.

Parameters

instance (*cms.models.pluginmodel.CMSPlugin* instance) – The instance of the plugin model.

Example:

```
def icon_src(self, instance):
    return static("cms/img/icons/plugins/link.png")
```

See also: *text_enabled*, *icon_alt()*

log_addition(*request*, *obj*, *bypass=None*)

Log that an object has been successfully added.

The default implementation creates an admin LogEntry object.

log_change(*request*, *obj*, *message*, *bypass=None*)

Log that an object has been successfully changed.

The default implementation creates an admin LogEntry object.

log_deletion(*request*, *obj*, *object_repr*, *bypass=None*)

Log that an object will be deleted. Note that this method must be called before the deletion.

The default implementation creates an admin LogEntry object.

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

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

Parameters

- **context** (*dict*) – The context with which the page is rendered.
- **instance** (*cms.models.pluginmodel.CMSPlugin* instance) – The instance of your plugin that is rendered.
- **placeholder** (*str*) – The name of the placeholder that is rendered.

Return type

dict or `django.template.Context`

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

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().render(context, instance, placeholder)
    ...
    return context
```

render_change_form(*request*, *context*, *add=False*, *change=False*, *form_url=''*, *obj=None*)

We just need the popup interface here

render_close_frame(*request: HttpRequest*, *obj:* CMSPlugin | *None*, *action: str* | *None* = *'edit'*, *extra_data: Context* | *None* = *None*, *extra_context: Context* | *None* = *None*) → HttpResponse

Renders the close frame for a CMS plugin in the Django admin interface.

This method is used to send information to the javascript frontend after an edit action has taken place. It allows the javascript frontend to update the structure and content on the edit endpoints.

Args:

request (HttpRequest): The HTTP request object. obj (CMSPlugin): The CMS plugin instance being rendered. action (Optional[str]): The action being performed on the plugin.
Defaults to "edit". Possible values are "add", "edit", "move", and "delete".

extra_data (Optional[Context]): Additional data to include in
the data bridge to the frontend. Defaults to None.

extra_context (Optional[Context]): Additional context to include in
the rendering. Defaults to None.

Returns:

HttpResponse: The rendered confirmation form as an HTTP response.

Raises:

ObjectDoesNotExist: If the parent plugin is a ghost plugin and the
plugin tree cannot be fetched.

Notes:

- Handles edge cases where the parent plugin is a ghost plugin.
- Constructs the plugin tree structure and gathers plugin-specific data for rendering.
- Includes messages from the request in the context.

`response_add`(*request*, *obj*, ***kwargs*)

Determine the HttpResponse for the add_view stage.

`response_change`(*request*, *obj*)

Determine the HttpResponse for the change_view stage.

`save_form`(*request*, *form*, *change*)

Given a ModelForm return an unsaved instance. `change` is True if the object is being changed, and False if it's being added.

`save_model`(*request*, *obj*, *form*, *change*)

Override original method, and add some attributes to obj This has to be made, because if the object is newly created, it must know where it lives.

`allow_children = 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*.

`allowed_models = None`

Plugin-level restriction: A list of valid models where this plugin can be added.

Each entry must be the dotted path to the model in the format `"app_label.modelname"`, e.g., `["cms.pagecontent", "myapp.mymodel"]`.

- If `None` (default): The plugin can be added to any model that has placeholders.
- If a list/tuple is provided: The plugin can only be added to models in this list.
- If an empty list `[]`: The plugin cannot be added to any model.

Note: This can be combined with the model's `allowed_plugins` attribute for fine-grained control. Both filters must pass for a plugin to be available on a model.

See also: Model's `allowed_plugins` attribute for model-level restrictions.

`cache = True`

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 = '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 = None`

A list of Plugin Class Names. If this is set, only plugins listed here can be added to this plugin. Entries may be glob patterns (e.g. `"Bootstrap*"` or `"*Link*"`), which are expanded against the names of all registered plugins. An empty list (`[]`), or a list of patterns that match no installed plugin, means that no plugins are allowed as children, whereas `None` (the default) means there is no restriction. As a special case, the string `"auto"` allows exactly those plugins that explicitly name this plugin in their *`parent_classes`* – i.e. the restriction is driven by the children opting in, rather than the parent listing them. See also: *`parent_classes`*.

`disable_child_plugins = False`

Disables *dragging* of child plugins in structure mode.

`form = None`

Custom form class to be used to edit this plugin.

is_local = True

The plugin does not modify the context or request and its rendering is not influenced by its parent plugins. Defaults to `True` unless *CMS_ALWAYS_REFRESH_CONTENT* is set to `True`.

is_slot = False

Marks this plugin as a *slot* — a structural container that is not directly editable by the user.

If `True`, the plugin will be rendered in structure mode normally, but double-clicking on it will not open the plugin edit dialog. The user will not have a direct way to change the plugin instance.

Moving or adding child plugins are not affected.

module = 'Generic'

Modules collect plugins of similar type

name = ''

Name of the plugin needs to be set in child classes

page_only = 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 `PlaceholderRelationFields`. See also: *child_classes*, *parent_classes*, *require_parent*.

Deprecated: Use allowed_models attribute instead (e.g., *allowed_models = ["cms.pagecontent"]*)

parent_classes = None

A list of the names of permissible parent classes for this plugin. Entries may be glob patterns (e.g. `"Bootstrap*"` or `"*Link*"`), which are expanded against the names of all registered plugins. An empty list (`[]`), or a list of patterns that match no installed plugin, means that the plugin allows no parents and can therefore only be added to a placeholder, whereas `None` (the default) means there is no restriction. See also: *child_classes*, *require_parent*.

render_plugin = True

If set to `False`, this plugin will not be rendered at all. If `True`, *render_template()* must also be defined. See also: *render_template*, *get_render_template()*.

render_template = 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 = 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*.

show_add_form = True

Determines if the add plugin modal is shown for this plugin (default: yes). Useful for plugins which have no fields to fill, or which have valid default values for *all* fields. If the plugin's form will not validate with the default values the add plugin modal is shown with the form errors.

system = False

This flag is used to determine whether a plugin is considered a "system" (internal) plugin — meaning, it is used by the cms itself to structure content or placeholders, not meant to be added, edited, or deleted directly by end users in the admin or frontend editor. When set to `True`, the cms treats that plugin as infrastructure, not user content (e.g djangocms-transfer).

text_enabled = False

This attribute controls whether your plugin will be usable (and rendered) in a text plugin. When you edit

a text plugin on a page, the plugin will show up in the *CMS Plugins* dropdown and can be configured and inserted. The output will even be previewed in the text editor.

Of course, not all plugins are usable in text plugins. Therefore the default of this attribute is `False`. If your plugin *is* usable in a text plugin:

1. set this to `True`
2. make sure your plugin provides its own *icon_alt()*, this will be used as a tooltip in

   the text-editor and comes in handy when you use multiple plugins in your text.

See also: *icon_alt()*, *icon_src()*.

class cms.plugin_base.**PluginMenuItem**(*name*, *url*, *data=None*, *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

class cms.models.pluginmodel.**CMSPlugin**(**args*, ***kwargs*)

The base class for a CMS plugin model. When defining a new custom plugin, you should store plugin-instance specific information on a subclass of this class. (An example for this would be to store the number of pictures to display in a gallery.)

Two restrictions apply when subclassing this to use in your own models:

1. Subclasses of CMSPlugin **cannot be further subclassed**
2. Subclasses of CMSPlugin cannot define a "text" field.

add_structureboard_classes(*self*)

Optional method on a *CMSPlugin* subclass. If defined, its return value is added as CSS classes to the `cms-draggable` container in the structure board. This can be used together with custom CSS to visually style a plugin or its children — for example, to mark deactivated plugins.

Return type

str

See *How to create Plugins* for a full example.

exception DoesNotExist

exception MultipleObjectsReturned

copy_relations(*old_instance*)

Handle copying of any relations attached to this plugin. Custom plugins have to do this themselves.

See also: *Handling Relations*, *post_copy()*.

Parameters

old_instance (*CMSPlugin* instance) – Source plugin instance

get_action_urls(*js_compat=True*)

Returns

dict of action urls for edit, add, delete, copy, and move plugin.

This method replaces the set of legacy methods *get_add_url*, ``get_edit_url`, *get_move_url*, *get_delete_url*, *get_copy_url*.

get_ancestors() → list[*CMSPlugin*]

Retrieve the list of ancestor plugins for the current plugin.

This method returns a list of ancestor plugins, starting from the root ancestor down to the immediate parent of the current plugin. If the current plugin has no parent, an empty list is returned.

Returns:

list: A list of ancestor plugins, ordered from the root ancestor to the immediate parent of the current plugin.

get_bound_plugin()

Returns an instance of the plugin model configured for this plugin type.

get_instance_icon_alt()

Get alt text for instance's icon

get_instance_icon_src()

Get src URL for instance's icon

get_plugin_instance(*admin=None*)

For a plugin instance (usually as a CMSPluginBase), this method returns the downcasted (i.e., correctly typed subclass of CMSPluginBase) instance and the plugin class

Returns

Tuple (instance, plugin)

instance: The instance AS THE APPROPRIATE SUBCLASS OF CMSPluginBase and not necessarily just 'self', which is often just a CMSPluginBase,

plugin: the associated plugin class instance (subclass of CMSPlugin)

notify_on_autoadd(*request*, *conf*)

Method called when we auto add this plugin via default_plugins in CMS_PLACEHOLDER_CONF.

Some specific plugins may have some special stuff to do when they are auto added.

notify_on_autoadd_children(*request*, *conf*, *children*)

Method called when we auto add children to this plugin via default_plugins/<plugin>/children in CMS_PLACEHOLDER_CONF.

Some specific plugins may have some special stuff to do when we add children to them. ie : TextPlugin must update its content to add HTML tags to be able to see his children in WYSIWYG.

post_copy(*old_instance*, *new_old_ziplist*)

Can (should) be overridden to handle the copying of plugins which contain children plugins after the original parent has been copied.

E.g., TextPlugins use this to correct the references in the text to child plugins. copied

refresh_from_db(**args*, ***kwargs*)

Reload field values from the database.

By default, the reloading happens from the database this instance was loaded from, or by the read router if this instance wasn't loaded from any database. The using parameter will override the default.

Fields can be used to specify which fields to reload. The fields should be an iterable of field attnames. If fields is None, then all non-deferred fields are reloaded.

When accessing deferred fields of an instance, the deferred loading of the field will call this method.

changed_date

django:django.db.models.DateTimeField: Datetime the plugin was last changed

child_class_restrictions = None

Request-scoped list of the plugin classes that may be added as children of this instance

child_plugin_instances = None

Request-scoped list of direct child plugin instances when rendering

creation_date

django:django.db.models.DateTimeField: Datetime the plugin was created

language

`django.db.models.CharField`: Language of the plugin

parent

`django.db.models.ForeignKey`: Parent plugin or `None` for plugins at root level in the placeholder

placeholder

`django.db.models.ForeignKey`: Placeholder the plugin belongs to

plugin_type

django:django.db.models.CharField: Plugin type (name of the class as string)

position

`django.db.models.SmallIntegerField`: Position (unique for placeholder and language) starting with 1 for the first plugin in the placeholder

class cms.plugin_pool.**PluginPool**

can_cache_globally(*plugin_class: type[*CMSPluginBase*]*) → bool

Check if the restrictions for a given plugin class can be cached globally.

This is the case if the plugin restrictions only depend on template and placeholder slot as described by the CMS_PLACEHOLDER_CONF setting.

Args:

plugin_class (CMSPluginBase): The plugin class for which to check if restrictions can be cached globally.

Returns:

bool: True if the restrictions can be cached globally, False otherwise.

expand_plugin_patterns(*patterns: tuple[str, ...]*) → frozenset[str]

Expand a tuple of plugin-name patterns into the set of matching plugin names.

Literal names are kept verbatim (so an unregistered literal simply matches no installed plugin, preserving historic behaviour). Glob patterns – `*`, `?` and `[seq]` – are matched case-sensitively against the names of all registered plugins.

The result is memoized; the cache is cleared whenever plugins are (un)registered (see `_clear_cached()`), so the registry is only scanned on the first use of a given pattern tuple. This keeps restriction evaluation – which is on the critical rendering path – cheap.

get_all_plugins_for_model(*model: type[Model]*) → list[type[*CMSPluginBase*]]

Retrieve all plugins that can be used to edit the given model.

This method applies two levels of filtering:

1. Plugin-level filtering (allowed_models on plugin): - If a plugin has allowed_models defined, the model must be in that list - If allowed_models is None, the plugin is available for all models
2. Model-level filtering (allowed_plugins on model): - If the model has allowed_plugins defined, only those plugins are returned - If allowed_plugins is None, all plugins (passing filter 1) are returned - If allowed_plugins is an empty list [], no plugins are returned

Args:
model: The Django model class to get plugins for
Returns:
List of plugin classes that can be used with this model

get_plugin(*name*) → type[*CMSPluginBase*]

Retrieve a plugin from the cache.

get_restrictions_cache(*request_cache: dict*, *instance:* CMSPluginBase, *obj: Model | None*) → defaultdict[str, dict]

Retrieve the restrictions cache for a given plugin instance.

This method checks if the plugin class can be cached globally. This is the case if the plugin restrictions only depend on template and placeholder slot as described by the CMS_PLACEHOLDER_CONF setting.

If it can, it retrieves the appropriate restrictions cache based on the template and slot of the plugin instance's placeholder. If not, it returns the (local) request cache which will be recalculated for each request.
Args:
request_cache (dict): The current request cache (only filled is non globally cacheable). instance (CMSPluginBase): The plugin instance for which to retrieve the restrictions cache. obj (Optional[models.Model]): The model instance associated with the plugin instance, if any.
Returns:
dict: The restrictions cache for the given plugin instance - or the cache valid for the request.

register_plugin(*plugin: type[*CMSPluginBase*]*) → type[*CMSPluginBase*]

Register the given plugin.

Static sanity checks is also performed.

If a plugin is already registered, this will raise PluginAlreadyRegistered.

resolve_plugin_patterns(*patterns: list[str] | None*) → list[str] | None

Resolve a `child_classes`/`parent_classes` value, expanding any glob patterns.

`None` (no restriction) is returned unchanged. A list without glob patterns is returned as-is, both to avoid overhead and to preserve its ordering. A list with at least one glob is expanded to the sorted list of matching registered plugin names.

Expansion happens *before* emptiness is interpreted, so a glob that matches nothing yields an empty list – i.e. "no plugins allowed", just like an explicit `[]`.

unregister_plugin(*plugin: type[*CMSPluginBase*]*) → None

Unregister the given plugin.

If the plugin isn't already registered, this will raise PluginNotRegistered.

validate_templates(*plugin: type[*CMSPluginBase*] | None = None*) → None

Verify that all plugins have a valid render template.

Plugins that have render_plugin=False and allow_children=False are always deemed valid.

Plugin utility functions

cms.utils.plugins.**assign_plugins**(*request*, *placeholders*, *template=None*, *lang=None*)

Fetch all plugins for the given `placeholders` and cast them down to the concrete instances in one query per type.

Parameters
- **request** – The current request.
- **placeholders** – An iterable of placeholder objects.

- **template** – (optional) The template object.
- **lang** – (optional) The language code.

This method assigns plugins to the given placeholders. It retrieves the plugins from the database based on the placeholders and the language. The plugins are then downcasted to their specific plugin types.

The plugins are split up by placeholder and stored in a dictionary where the key is the placeholder ID and the value is a list of plugins.

For each placeholder, if there are plugins assigned to it, the plugins are organized as a layered tree structure. Otherwise, an empty list is assigned.

The list of all plugins for each placeholder is stored in the *_all_plugins_cache* attribute of the placeholder, while the list of root plugins is stored in the *_plugins_cache* attribute

cms.utils.plugins.**copy_plugins_to_placeholder**(*plugins*, *placeholder*, *language=None*, *root_plugin=None*, *start_positions=None*)

Copies an iterable of plugins to a placeholder

Parameters

- **plugins** (*iterable*) – Plugins to be copied
- **placeholder** (*cms.models.pluginmodel.CMSPlugin* instance) – Target placeholder
- **language** (*str*) – target language (if no root plugin is given)
- **root_plugin**
- **start_positions** (*int*) – Cache for start positions by language

The logic of this method is the following:

1. Get bound plugins for each source plugin
2. Get the parent plugin (if it exists)
3. then get a copy of the source plugin instance
4. Set the id/pk to None to it the id of the generic plugin instance above; this will effectively change the generic plugin created above into a concrete one
5. find the position in the new placeholder
6. save the concrete plugin (which creates a new plugin in the database)
7. trigger the copy relations
8. return the plugin ids

cms.utils.plugins.**downcast_plugins**(*plugins: Iterable[*CMSPlugin*]*, *placeholders: list | None = None*, *select_placeholder: bool = False*, *request: HttpRequest | None = None*) → Iterable[*CMSPlugin*]

Downcasts the given list of plugins to their respective classes. Ignores any plugins that are not available.

Parameters

- **plugins** (*List[CMSPlugin]*) – List of plugins to downcast.
- **placeholders** (*Optional[List[Placeholder]]*) – List of placeholders associated with the plugins.
- **select_placeholder** (*bool*) – If True, select_related the plugin queryset with placeholder.
- **request** (*Optional[HttpRequest]*) – The current request.

Returns

Generator that yields the downcasted plugins.

Return type

Generator[*CMSPlugin*, None, None]

cms.utils.plugins.**get_bound_plugins**(*plugins*)

Get the bound plugins by downcasting the plugins to their respective classes. Raises a KeyError if the plugin type is not available.

Creates a map of plugin types and their corresponding plugin IDs for later use in downcasting. Then, retrieves the plugin instances from the plugin model using the mapped plugin IDs. Finally, iterates over the plugins and

yields the downcasted versions if they have a valid parent. Does not affect caching.

Parameters

plugins (*List[CMSPlugin]*) – List of CMSPlugin instances.

Returns

Generator that yields the downcasted plugins.

Return type

Generator[*CMSPlugin*, None, None]

Example:

```
plugins = [plugin_instance1, plugin_instance2]
for bound_plugin in get_bound_plugins(plugins):
    # Do something with the bound_plugin
    pass
```

cms.utils.plugins.**get_plugin_class**(*plugin_type: str*) → type[*CMSPluginBase*]

Returns the plugin class for a given plugin_type (str)

cms.utils.plugins.**get_plugin_model**(*plugin_type: str*) → *CMSPlugin*

Returns the plugin model class for a given plugin_type (str)

cms.utils.plugins.**get_plugin_restrictions**(*plugin*, *page=None*, *restrictions_cache=None*)

cms.utils.plugins.**get_plugins**(*request*, *placeholder*, *template*, *lang=None*)

Get a list of plugins for a placeholder in a specified template. Respects the placeholder's cache.

Parameters

- **request** – (HttpRequest) The HTTP request object.
- **placeholder** – (Placeholder) The placeholder object for which to retrieve plugins.
- **template** – (Template) The template object in which the placeholder resides (not used).
- **lang** – (str, optional) The language code for localization. Defaults to None.

Returns:

list: A list of plugins for the specified placeholder in the template.

Raises:

None.

Examples:

```
# Get plugins for a placeholder in a template
plugins = get_plugins(request, placeholder, template)

# Get plugins for a placeholder in a template with specific language
plugins = get_plugins(request, placeholder, template, lang="en")
```

cms.utils.plugins.**get_plugins_as_layered_tree**(*plugins*)

Given an iterable of plugins ordered by position, returns a deque of root plugins with their respective children set in the child_plugin_instances attribute.

cms.utils.plugins.**has_reached_plugin_limit**(*placeholder*, *plugin_type*, *language*, *template=None*)

Checks if the global maximum limit for plugins in a placeholder has been reached. If not then it checks if it has reached its maximum plugin_type limit.

Parameters: - placeholder: The placeholder object to check the limit for. - plugin_type: The type of plugin to check the limit for. - language: The language code for the plugins. - template: The template object for the placeholder. Optional.

Returns: - False if the limit has not been reached.

Raises: - PluginLimitReached: If the limit has been reached for the placeholder.

Sitemaps

class cms.sitemaps.cms_sitemap.**CMSSitemap**

Bases: Sitemap

Sitemap for django CMS pages limited to public, non-redirecting, and anonymously accessible content on the current Site.

Targets the current Site and respects the project's public languages. Pages are annotated with the latest `PageContent.changed_date` and exposes it via `lastmod()`.

Enable the sitemap framework and register the sitemap in your URL configuration.

```
# settings.py
INSTALLED_APPS = [
    # ...
    "django.contrib.sitemaps",
]
```

```
# urls.py
from django.contrib.sitemaps import views as sitemap_views
from cms.sitemaps.cms_sitemap import CMSSitemap

sitemaps = {
    "pages": CMSSitemap,
}

urlpatterns = [
    path("sitemap.xml", sitemap_views.sitemap, {"sitemaps": sitemaps}, name="sitemap
↪"),
]
```

items() → QuerySet

Items are PageUrl instances for the current Site, filtered to only include those that are:

- In a public language for the Site
- Not redirects
- Not login_required

The site is taken from the `SITE_ID` setting or identified from the current request using the Sites framework. (A custom site middelware is ignored by Django's sitemap framework.)

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

django CMS provides two template tags for placeholders:

- *`placeholder`* **declares and renders** a placeholder inside a template. Use it for page contents, and (since django CMS 5.1) for any apphooked model that uses a *`PlaceholderRelationField`* — see *Using placeholders on an apphooked page*. Because *`placeholder`* also *declares* the placeholder, django CMS can auto-detect it and

expose it to editors. In a *structure template* the `placeholder` tag is used for the declaration only; the actual rendering happens in a separate user-facing template via `render_placeholder`.

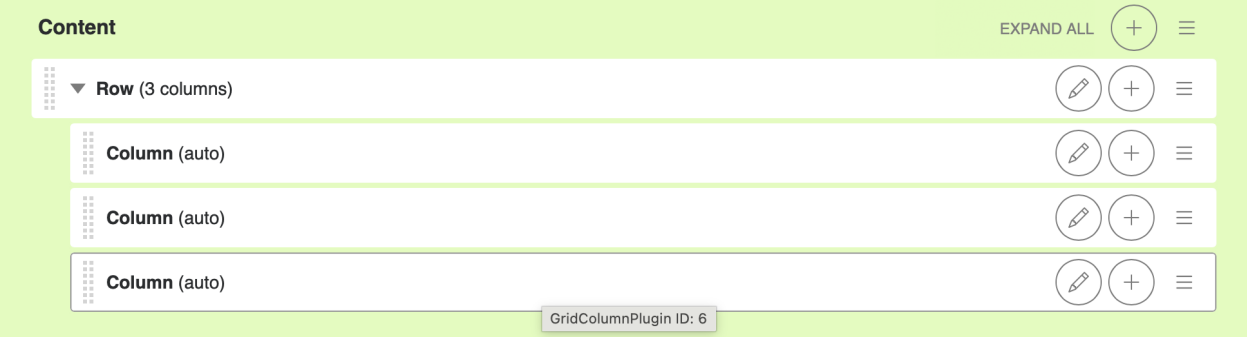
- `render_placeholder` **only renders** an existing placeholder; it does not declare one. It takes a `Placeholder` instance as a parameter, which the model must provide — typically by retrieving it from a `PlaceholderRelationField` via `get_placeholder_from_slot()`. Use it for existing apphooks that already pass a placeholder instance, and for models that expose pre-defined placeholders this way.

placeholder

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>.

Changed in version 5.1: The *placeholder* tag can also be used in templates rendered by apphook views. It resolves against the current page's content by default, or against the toolbar object if the apphook view sets one. See *Using placeholders on an apphooked page*.

static_placeholder

Changed in version 4.0.

The `static_placeholder` template tag does **not** work with django CMS 4. It will be removed in a future version.

Note

As a replacement use django CMS Alias instead. Once installed use `{% load djangocms_alias_tags %}` and `{% static_alias "footer" %}` as a replacement for `static_placeholder`

In connection with django CMS Versioning you can better manage versions of page parts that appear at several instances on your pages.

render_placeholder

`{% render_placeholder %}` is used if you have a *PlaceholderRelationField* on your own model and want to render one of its placeholders in the template.

The *render_placeholder* tag takes the following parameters:

- *Placeholder* 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' %}
```

Added 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.

See *How to use placeholders outside the CMS* or *PlaceholderRelationField* on how to get a specific placeholder instance.

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.

Added 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 %}
```

Added 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>
```

Added 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.

Added 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>
```

(continues on next page)

(continued from previous page)

```
  <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 %}
```

Added 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 behaviour, 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.

Added 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 behaviour, 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.

Added 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 behaviour, 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.

Added 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 behaviour, 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.

Added 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 *Serving content in multiple languages*.

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 *Serving content in multiple languages*.

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.

Toolbar

The toolbar can contain various items, some of which in turn can contain other items. These items are represented by the classes listed in *`cms.toolbar.items`*, and created using the various APIs described below.

Do not instantiate these classes manually

These classes are described here for reference purposes only. It is strongly recommended that you do not create instances yourself, but use the methods listed here.

Classes and methods

Common parameters (`key`, `verbose_name`, `position`, `on_close`, `disabled`, `active`) and options are described at the end of this document.

class cms.toolbar.toolbar.**BaseToolbar**

Bases: *ToolbarAPIMixin*

add_ajax_item(*name*, *action*, *active=False*, *disabled=False*, *extra_classes=None*, *data=None*, *question=None*, *side=<object object>*, *position=None*, *on_success=None*, *method='POST'*)

Adds *AjaxItem* that sends a request to `action` with `data`, and returns it. `data` should be `None` or a dictionary. By default a `POST` request is sent; pass `method` to use a different HTTP method. For unsafe methods (anything other than `GET`, `HEAD`, `OPTIONS` and `TRACE`) the CSRF token is automatically added to the item.

If a string is provided for `question`, it will be presented to the user to allow confirmation before the request is sent.

Once the request succeeds, the response is inspected:

- If `on_success` is given, the browser navigates to that URL. The special values `REFRESH_PAGE` reloads the current page and `"FOLLOW_REDIRECT"` redirects to the `url` returned in the (JSON) response instead.
- Otherwise, if the response carries a *data bridge* – either an HTML document containing a `<script id="data-bridge">` element or a JSON object with an `action` key – the data bridge is evaluated and the structure board is updated in place, without a full page reload.
- If neither applies, the page is reloaded.

Added in version 5.1: Evaluating a *data bridge* returned by the AJAX response, updating the structure board in place instead of reloading the page.

add_item(*item*, *position=None*)

Adds an item (which must be a subclass of *BaseItem*), and returns it. This is a low-level API, and you should always use one of the built-in object-specific methods to add items in preference if possible, using this method **only** for custom item classes.

add_link_item(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *side=<object object>*, *position=None*)

Adds a *LinkItem* that opens `url`, and returns it.

add_modal_item(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *on_close='REFRESH_PAGE'*, *side=<object object>*, *position=None*)

Similar to *add_sideframe_item()*, but adds a *ModalItem* that opens the `url` in a modal dialog instead of the sideframe, and returns it.

add_sideframe_item(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *on_close=None*, *side=<object object>*, *position=None*)

Adds a *SideframeItem* that opens `url` in the sideframe and returns it.

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()*. The return value of this method is safe to use as the *position* argument of the various APIs to add items.

find_items(*item_type*, ***attributes*)

Returns a list of *ItemSearchResult* objects matching all items of `item_type` (e.g. `LinkItem`).

Items whose attributes are lazy (e.g. a `gettext_lazy` `name`) are matched by their *source* string, not their translation. Search for an item by the original (untranslated, usually English) string – e.g. `find_items(LinkItem, name="Save")` – so the lookup works regardless of the active language. Passing the lazy itself or a translated string is not reliable across languages.

This source matching only works for single-argument lazies such as `_("Save")`. Composite lazy values – e.g. `format_lazy("Save {}", model_name)` – have no single source string and are matched by their

resolved (translated) value, so they cannot be looked up language-independently. Give such items a stable `identifier` and search by that instead.

get_item_count()

Returns the number of items in the menu.

content_mode_active

`True` if content mode is active.

edit_mode_active

`True` if the structure board editing mode is active.

preview_mode_active

`True` if preview mode is active.

watch_models = []

Property; a list of models that the toolbar *watches for URL changes*, so it can redirect to the new URL on saving.

class cms.toolbar.toolbar.**CMSToolbarBase**(*request*, *request_path=None*, *_async=False*)

Bases: *BaseToolbar*

The toolbar is an instance of the *CMSToolbar* class. This should not be confused with the *CMSToolbar*, the base class for *toolbar modifier classes* in other applications, that add items to and otherwise manipulates the toolbar.

It is strongly recommended that you **only** interact with the toolbar in your own code via:

1. the APIs documented here
2. toolbar modifier classes based on *CMSToolbar*

Several of the following methods to create and add objects other objects to the toolbar are inherited from *ToolbarAPIMixin*.

add_ajax_item(*name*, *action*, *active=False*, *disabled=False*, *extra_classes=None*, *data=None*, *question=None*, *side=<object object>*, *position=None*, *on_success=None*, *method='POST'*)

Adds *AjaxItem* that sends a request to `action` with `data`, and returns it. `data` should be `None` or a dictionary. By default a `POST` request is sent; pass `method` to use a different HTTP method. For unsafe methods (anything other than `GET`, `HEAD`, `OPTIONS` and `TRACE`) the CSRF token is automatically added to the item.

If a string is provided for `question`, it will be presented to the user to allow confirmation before the request is sent.

Once the request succeeds, the response is inspected:

- If `on_success` is given, the browser navigates to that URL. The special values `REFRESH_PAGE` reloads the current page and `"FOLLOW_REDIRECT"` redirects to the `url` returned in the (JSON) response instead.
- Otherwise, if the response carries a *data bridge* – either an HTML document containing a `<script id="data-bridge">` element or a JSON object with an `action` key – the data bridge is evaluated and the structure board is updated in place, without a full page reload.
- If neither applies, the page is reloaded.

Added in version 5.1: Evaluating a *data bridge* returned by the AJAX response, updating the structure board in place instead of reloading the page.

add_button(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *extra_wrapper_classes=None*, *side=<object object>*, *position=None*)

Adds a *Button* to the toolbar.

add_button_list(*identifier=None*, *extra_classes=None*, *side=<object object>*, *position=None*)

Adds an (empty) *ButtonList* to the toolbar and returns it.

add_item(*item*, *position=None*)

Adds an item (which must be a subclass of *BaseItem*), and returns it. This is a low-level API, and you should always use one of the built-in object-specific methods to add items in preference if possible, using this method **only** for custom item classes.

add_link_item(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *side=<object object>*, *position=None*)

Adds a *LinkItem* that opens `url`, and returns it.

add_modal_button(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *extra_wrapper_classes=None*, *side=<object object>*, *position=None*, *on_close='REFRESH_PAGE'*)

Adds a *ModalButton* to the toolbar.

add_modal_item(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *on_close='REFRESH_PAGE'*, *side=<object object>*, *position=None*)

Similar to *add_sideframe_item()*, but adds a *ModalItem* that opens the `url` in a modal dialog instead of the sideframe, and returns it.

add_sideframe_button(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *extra_wrapper_classes=None*, *side=<object object>*, *position=None*, *on_close=None*)

Adds a *SideframeButton* to the toolbar.

add_sideframe_item(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *on_close=None*, *side=<object object>*, *position=None*)

Adds a *SideframeItem* that opens `url` in the sideframe and returns it.

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()*. The return value of this method is safe to use as the *position* argument of the various APIs to add items.

find_items(*item_type*, ***attributes*)

Returns a list of *ItemSearchResult* objects matching all items of `item_type` (e.g. `LinkItem`).

Items whose attributes are lazy (e.g. a `gettext_lazy` `name`) are matched by their *source* string, not their translation. Search for an item by the original (untranslated, usually English) string – e.g. `find_items(LinkItem, name="Save")` – so the lookup works regardless of the active language. Passing the lazy itself or a translated string is not reliable across languages.

This source matching only works for single-argument lazies such as `_("Save")`. Composite lazy values – e.g. `format_lazy("Save {}", model_name)` – have no single source string and are matched by their *resolved* (translated) value, so they cannot be looked up language-independently. Give such items a stable `identifier` and search by that instead.

get_item_count()

Returns the number of items in the menu.

get_menu(*key*, *verbose_name=None*, *side=<object object>*, *position=None*)

Will return the `Menu` identified with *key*, or `None`.

get_object()

Returns the object currently associated with the toolbar.

This returns the object that was previously set using *set_object()*, or `None` if no object has been associated with the toolbar.

Returns
The object associated with the toolbar, or None
Return type
django.db.models.Model or None

`get_or_create_menu`(*key*, *verbose_name=None*, *disabled=False*, *side=<object object>*, *position=None*)

If a *Menu* with *key* already exists, this method will return that menu. Otherwise it will create a menu with the `key` identifier.

`populate`()

Populates the toolbar with the CMS items.

`set_object`(*obj*)

Associates an object with the toolbar.

Sets the toolbar's object if one has not already been set. This object is typically a Django model instance that the toolbar should operate on, such as a *PageContent* object or any other model that supports editable placeholders through a *PlaceholderRelationField*.

The object is used by other toolbar methods like `get_object_edit_url()`, `get_object_preview_url()`, and `get_object_structure_url()` to generate appropriate URLs for the object.

Parameters
obj (*django.db.models.Model*) – The object to associate with the toolbar

`content_mode_active`

`True` if content mode is active.

`edit_mode_active`

`True` if the structure board editing mode is active.

`preview_mode_active`

`True` if preview mode is active.

`watch_models = []`

Property; a list of models that the toolbar *watches for URL changes*, so it can redirect to the new URL on saving.

`class` cms.toolbar.toolbar.`CMSToolbar`(*request*, *request_path=None*, *_async=False*)

Bases: *CMSToolbarBase*

CMSToolbarBase including toolbar mixins from extensions to toolbar

`add_ajax_item`(*name*, *action*, *active=False*, *disabled=False*, *extra_classes=None*, *data=None*, *question=None*, *side=<object object>*, *position=None*, *on_success=None*, *method='POST'*)

Adds *AjaxItem* that sends a request to `action` with `data`, and returns it. `data` should be `None` or a dictionary. By default a `POST` request is sent; pass `method` to use a different HTTP method. For unsafe methods (anything other than `GET`, `HEAD`, `OPTIONS` and `TRACE`) the CSRF token is automatically added to the item.

If a string is provided for `question`, it will be presented to the user to allow confirmation before the request is sent.

Once the request succeeds, the response is inspected:

- If `on_success` is given, the browser navigates to that URL. The special values `REFRESH_PAGE` reloads the current page and `"FOLLOW_REDIRECT"` redirects to the `url` returned in the (JSON) response instead.

- Otherwise, if the response carries a *data bridge* – either an HTML document containing a `<script id="data-bridge">` element or a JSON object with an `action` key – the data bridge is evaluated and the structure board is updated in place, without a full page reload.
- If neither applies, the page is reloaded.

Added in version 5.1: Evaluating a *data bridge* returned by the AJAX response, updating the structure board in place instead of reloading the page.

add_button(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *extra_wrapper_classes=None*, *side=<object object>*, *position=None*)

Adds a *Button* to the toolbar.

add_button_list(*identifier=None*, *extra_classes=None*, *side=<object object>*, *position=None*)

Adds an (empty) *ButtonList* to the toolbar and returns it.

add_item(*item*, *position=None*)

Adds an item (which must be a subclass of *BaseItem*), and returns it. This is a low-level API, and you should always use one of the built-in object-specific methods to add items in preference if possible, using this method **only** for custom item classes.

add_link_item(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *side=<object object>*, *position=None*)

Adds a *LinkItem* that opens `url`, and returns it.

add_modal_button(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *extra_wrapper_classes=None*, *side=<object object>*, *position=None*, *on_close='REFRESH_PAGE'*)

Adds a *ModalButton* to the toolbar.

add_modal_item(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *on_close='REFRESH_PAGE'*, *side=<object object>*, *position=None*)

Similar to *add_sideframe_item()*, but adds a *ModalItem* that opens the `url` in a modal dialog instead of the sideframe, and returns it.

add_sideframe_button(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *extra_wrapper_classes=None*, *side=<object object>*, *position=None*, *on_close=None*)

Adds a *SideframeButton* to the toolbar.

add_sideframe_item(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *on_close=None*, *side=<object object>*, *position=None*)

Adds a *SideframeItem* that opens `url` in the sideframe and returns it.

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()*. The return value of this method is safe to use as the *position* argument of the various APIs to add items.

find_items(*item_type*, ***attributes*)

Returns a list of *ItemSearchResult* objects matching all items of `item_type` (e.g. `LinkItem`).

Items whose attributes are lazy (e.g. a `gettext_lazy` `name`) are matched by their *source* string, not their translation. Search for an item by the original (untranslated, usually English) string – e.g. `find_items(LinkItem, name="Save")` – so the lookup works regardless of the active language. Passing the lazy itself or a translated string is not reliable across languages.

This source matching only works for single-argument lazies such as `_("Save")`. Composite lazy values – e.g. `format_lazy("Save {}", model_name)` – have no single source string and are matched by their

resolved (translated) value, so they cannot be looked up language-independently. Give such items a stable `identifier` and search by that instead.

`get_item_count`()

Returns the number of items in the menu.

`get_menu`(*key*, *verbose_name=None*, *side=<object object>*, *position=None*)

Will return the `Menu` identified with *key*, or `None`.

`get_object`()

Returns the object currently associated with the toolbar.

This returns the object that was previously set using *set_object()*, or `None` if no object has been associated with the toolbar.

Returns

The object associated with the toolbar, or None

Return type

django.db.models.Model or None

`get_or_create_menu`(*key*, *verbose_name=None*, *disabled=False*, *side=<object object>*, *position=None*)

If a *Menu* with *key* already exists, this method will return that menu. Otherwise it will create a menu with the `key` identifier.

`populate`()

Populates the toolbar with the CMS items.

`set_object`(*obj*)

Associates an object with the toolbar.

Sets the toolbar's object if one has not already been set. This object is typically a Django model instance that the toolbar should operate on, such as a *PageContent* object or any other model that supports editable placeholders through a *PlaceholderRelationField*.

The object is used by other toolbar methods like `get_object_edit_url()`, `get_object_preview_url()`, and `get_object_structure_url()` to generate appropriate URLs for the object.

Parameters

obj (*django.db.models.Model*) – The object to associate with the toolbar

`content_mode_active`

`True` if content mode is active.

`edit_mode_active`

`True` if the structure board editing mode is active.

`preview_mode_active`

`True` if preview mode is active.

`watch_models = []`

Property; a list of models that the toolbar *watches for URL changes*, so it can redirect to the new URL on saving.

`class` cms.toolbar.items.`FrameItem`(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *on_close=None*, *side=<object object>*)

Bases: *BaseItem*

Base class for *ModalItem* and *SideframeItem*. Frame items have three dots besides their name indicating that some frame or dialog will open when selected.

get_context()

Returns the context (as dictionary) for this item.

class cms.toolbar.items.**Menu**(*name*, *csrf_token*, *disabled=False*, *side=<object object>*)

Bases: *SubMenu*

Provides a menu in the toolbar. Use a *CMSToolbar.get_or_create_menu* method to create a `Menu` instance. Can be added to *CMSToolbar*.

add_ajax_item(*name*, *action*, *active=False*, *disabled=False*, *extra_classes=None*, *data=None*, *question=None*, *side=<object object>*, *position=None*, *on_success=None*, *method='POST'*)

Adds *AjaxItem* that sends a request to `action` with `data`, and returns it. `data` should be `None` or a dictionary. By default a `POST` request is sent; pass `method` to use a different HTTP method. For unsafe methods (anything other than `GET`, `HEAD`, `OPTIONS` and `TRACE`) the CSRF token is automatically added to the item.

If a string is provided for `question`, it will be presented to the user to allow confirmation before the request is sent.

Once the request succeeds, the response is inspected:

- If `on_success` is given, the browser navigates to that URL. The special values `REFRESH_PAGE` reloads the current page and `"FOLLOW_REDIRECT"` redirects to the `url` returned in the (JSON) response instead.
- Otherwise, if the response carries a *data bridge* – either an HTML document containing a `<script id="data-bridge">` element or a JSON object with an `action` key – the data bridge is evaluated and the structure board is updated in place, without a full page reload.
- If neither applies, the page is reloaded.

Added in version 5.1: Evaluating a *data bridge* returned by the AJAX response, updating the structure board in place instead of reloading the page.

add_break(*identifier=None*, *position=None*)

Adds a visual break in the menu, at *position*, and returns it. `identifier` may be used to make this item searchable.

add_item(*item*, *position=None*)

Adds an item (which must be a subclass of *BaseItem*), and returns it. This is a low-level API, and you should always use one of the built-in object-specific methods to add items in preference if possible, using this method **only** for custom item classes.

add_link_item(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *side=<object object>*, *position=None*)

Adds a *LinkItem* that opens `url`, and returns it.

add_modal_item(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *on_close='REFRESH_PAGE'*, *side=<object object>*, *position=None*)

Similar to *add_sideframe_item()*, but adds a *ModalItem* that opens the `url` in a modal dialog instead of the sideframe, and returns it.

add_sideframe_item(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *on_close=None*, *side=<object object>*, *position=None*)

Adds a *SideframeItem* that opens `url` in the sideframe and returns it.

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()*. The return value of this method is safe to use as the *position* argument of the various APIs to add items.

find_items(*item_type*, ***attributes*)

Returns a list of *ItemSearchResult* objects matching all items of `item_type` (e.g. `LinkItem`).

Items whose attributes are lazy (e.g. a `gettext_lazy name`) are matched by their *source* string, not their translation. Search for an item by the original (untranslated, usually English) string – e.g. `find_items(LinkItem, name="Save")` – so the lookup works regardless of the active language. Passing the lazy itself or a translated string is not reliable across languages.

This source matching only works for single-argument lazies such as `_("Save")`. Composite lazy values – e.g. `format_lazy("Save {}", model_name)` – have no single source string and are matched by their *resolved* (translated) value, so they cannot be looked up language-independently. Give such items a stable `identifier` and search by that instead.

get_context()

Returns the context (as dictionary) for this item.

get_item_count()

Returns the number of items in the menu.

get_or_create_menu(*key*, *verbose_name*, *disabled=False*, *side=<object object>*, *position=None*)

Adds a new sub-menu, at *position*, and returns a *SubMenu*.

render()

Renders the item and returns it as a string. By default, calls *get_context()* and renders *template* with the context returned.

template = 'cms/toolbar/items/menu.html'

Must be set by subclasses and point to a Django template

class cms.toolbar.items.**SubMenu**(*name*, *csrf_token*, *disabled=False*, *side=<object object>*)

Bases: *ToolbarAPIMixin*, *BaseItem*

A child of a *Menu*. Use a *Menu.get_or_create_menu* method to create a `SubMenu` instance. Can be added to `Menu`.

add_ajax_item(*name*, *action*, *active=False*, *disabled=False*, *extra_classes=None*, *data=None*, *question=None*, *side=<object object>*, *position=None*, *on_success=None*, *method='POST'*)

Adds *AjaxItem* that sends a request to `action` with `data`, and returns it. `data` should be `None` or a dictionary. By default a `POST` request is sent; pass `method` to use a different HTTP method. For unsafe methods (anything other than `GET`, `HEAD`, `OPTIONS` and `TRACE`) the CSRF token is automatically added to the item.

If a string is provided for `question`, it will be presented to the user to allow confirmation before the request is sent.

Once the request succeeds, the response is inspected:

- If `on_success` is given, the browser navigates to that URL. The special values `REFRESH_PAGE` reloads the current page and `"FOLLOW_REDIRECT"` redirects to the `url` returned in the (JSON) response instead.
- Otherwise, if the response carries a *data bridge* – either an HTML document containing a `<script id="data-bridge">` element or a JSON object with an `action` key – the data bridge is evaluated and the structure board is updated in place, without a full page reload.
- If neither applies, the page is reloaded.

Added in version 5.1: Evaluating a *data bridge* returned by the AJAX response, updating the structure board in place instead of reloading the page.

add_break(*identifier=None*, *position=None*)

Adds a visual break in the menu, at *position*, and returns it. `identifier` may be used to make this item searchable.

add_item(*item*, *position=None*)

Adds an item (which must be a subclass of *BaseItem*), and returns it. This is a low-level API, and you should always use one of the built-in object-specific methods to add items in preference if possible, using this method **only** for custom item classes.

add_link_item(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *side=<object object>*, *position=None*)

Adds a *LinkItem* that opens `url`, and returns it.

add_modal_item(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *on_close='REFRESH_PAGE'*, *side=<object object>*, *position=None*)

Similar to *add_sideframe_item()*, but adds a *ModalItem* that opens the `url` in a modal dialog instead of the sideframe, and returns it.

add_sideframe_item(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *on_close=None*, *side=<object object>*, *position=None*)

Adds a *SideframeItem* that opens `url` in the sideframe and returns it.

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()*. The return value of this method is safe to use as the *position* argument of the various APIs to add items.

find_items(*item_type*, ***attributes*)

Returns a list of *ItemSearchResult* objects matching all items of `item_type` (e.g. `LinkItem`).

Items whose attributes are lazy (e.g. a `gettext_lazy` `name`) are matched by their *source* string, not their translation. Search for an item by the original (untranslated, usually English) string – e.g. `find_items(LinkItem, name="Save")` – so the lookup works regardless of the active language. Passing the lazy itself or a translated string is not reliable across languages.

This source matching only works for single-argument lazies such as `_("Save")`. Composite lazy values – e.g. `format_lazy("Save {}", model_name)` – have no single source string and are matched by their *resolved* (translated) value, so they cannot be looked up language-independently. Give such items a stable `identifier` and search by that instead.

get_context()

Returns the context (as dictionary) for this item.

get_item_count()

Returns the number of items in the menu.

render()

Renders the item and returns it as a string. By default, calls *get_context()* and renders *template* with the context returned.

template = 'cms/toolbar/items/menu.html'

Must be set by subclasses and point to a Django template

class cms.toolbar.items.**LinkItem**(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *side=<object object>*)

Bases: *BaseItem*

Sends a GET request. Use an *add_link_item* method to create a `LinkItem` instance. Can be added to *CMSToolbar*, *Menu*, *SubMenu*.

get_context()

Returns the context (as dictionary) for this item.

template = 'cms/toolbar/items/item_link.html'

Must be set by subclasses and point to a Django template

class cms.toolbar.items.**SideframeItem**(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *on_close=None*, *side=<object object>*)

Bases: *FrameItem*

Sends a GET request; loads response in a sideframe. Use an *add_sideframe_item* method to create a `SideframeItem` instance. Can be added to *CMSToolbar*, *Menu*, *SubMenu*.

template = 'cms/toolbar/items/item_sideframe.html'

Must be set by subclasses and point to a Django template

class cms.toolbar.items.**ModalItem**(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *on_close=None*, *side=<object object>*)

Bases: *FrameItem*

Sends a GET request; loads response in a modal window. Use an *add_modal_item* method to create a `ModalItem` instance. Can be added to *CMSToolbar*, *Menu*, *SubMenu*.

template = 'cms/toolbar/items/item_modal.html'

Must be set by subclasses and point to a Django template

class cms.toolbar.items.**AjaxItem**(*name*, *action*, *csrf_token*, *data=None*, *active=False*, *disabled=False*, *extra_classes=None*, *question=None*, *side=<object object>*, *on_success=None*, *method='POST'*)

Bases: *BaseItem*

Sends a POST request. Use an *add_ajax_item* method to create a `AjaxItem` instance. Can be added to *CMSToolbar*, *Menu*, *SubMenu*.

get_context()

Returns the context (as dictionary) for this item.

template = 'cms/toolbar/items/item_ajax.html'

Must be set by subclasses and point to a Django template

class cms.toolbar.items.**Break**(*identifier=None*)

Bases: *BaseItem*

A visual break in a menu. Use an *add_break* method to create a `Break` instance. Can be added to *Menu*, *SubMenu*.

template = 'cms/toolbar/items/break.html'

Must be set by subclasses and point to a Django template

class cms.toolbar.items.**ButtonList**(*identifier=None*, *extra_classes=None*, *side=<object object>*)

Bases: *BaseItem*

A visually-connected list of one or more buttons. Use an *add_button_list()* method to create a `ButtonList` instance. Can be added to *CMSToolbar*.

add_button(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*)

Adds a *Button* to the list of buttons and returns it.

add_modal_button(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *on_close='REFRESH_PAGE'*)

Adds a *ModalButton* to the button list and returns it.

add_sideframe_button(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *on_close=None*)

Adds a *SideframeButton* to the button list and returns it.

get_buttons()

Yields all buttons in the button list

get_context()

Returns the context (as dictionary) for this item.

template = 'cms/toolbar/items/button_list.html'

Must be set by subclasses and point to a Django template

class cms.toolbar.items.**Button**(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*)

Bases: *BaseButton*

Sends a GET request. Use a *CMSToolbar.add_button* or *ButtonList.add_button()* method to create a Button instance. Can be added to *CMSToolbar*, *ButtonList*.

class cms.toolbar.items.**SideframeButton**(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *on_close=None*)

Bases: *ModalButton*

Sends a GET request. Use a *CMSToolbar.add_sideframe_button* or *ButtonList.add_sideframe_button()* method to create a SideframeButton instance. Can be added to *CMSToolbar*, *ButtonList*.

class cms.toolbar.items.**ModalButton**(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *on_close=None*)

Bases: *Button*

Sends a GET request. Use a *CMSToolbar.add_modal_button* or *ButtonList.add_modal_button()* method to create a ModalButton instance. Can be added to *CMSToolbar*, *ButtonList*.

class cms.toolbar.items.**BaseItem**(*side=<object object>*)

Bases: object

All toolbar items inherit from BaseItem. If you need to create a custom toolbar item, subclass BaseItem.

get_context()

Returns the context (as dictionary) for this item.

render()

Renders the item and returns it as a string. By default, calls *get_context()* and renders *template* with the context returned.

template = None

Must be set by subclasses and point to a Django template

class cms.toolbar.items.**ToolbarAPIMixin**

Provides APIs used by *CMSToolbar* and *Menu*.

add_ajax_item(*name*, *action*, *active=False*, *disabled=False*, *extra_classes=None*, *data=None*, *question=None*, *side=<object object>*, *position=None*, *on_success=None*, *method='POST'*)

Adds *AjaxItem* that sends a request to action with data, and returns it. data should be None or a dictionary. By default a POST request is sent; pass method to use a different HTTP method. For unsafe methods (anything other than GET, HEAD, OPTIONS and TRACE) the CSRF token is automatically added to the item.

If a string is provided for `question`, it will be presented to the user to allow confirmation before the request is sent.

Once the request succeeds, the response is inspected:

- If `on_success` is given, the browser navigates to that URL. The special values `REFRESH_PAGE` reloads the current page and `"FOLLOW_REDIRECT"` redirects to the `url` returned in the (JSON) response instead.
- Otherwise, if the response carries a *data bridge* – either an HTML document containing a `<script id="data-bridge">` element or a JSON object with an `action` key – the data bridge is evaluated and the structure board is updated in place, without a full page reload.
- If neither applies, the page is reloaded.

Added in version 5.1: Evaluating a *data bridge* returned by the AJAX response, updating the structure board in place instead of reloading the page.

add_item(*item*, *position=None*)

Adds an item (which must be a subclass of *BaseItem*), and returns it. This is a low-level API, and you should always use one of the built-in object-specific methods to add items in preference if possible, using this method **only** for custom item classes.

add_link_item(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *side=<object object>*, *position=None*)

Adds a *LinkItem* that opens `url`, and returns it.

add_modal_item(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *on_close='REFRESH_PAGE'*, *side=<object object>*, *position=None*)

Similar to *add_sideframe_item()*, but adds a *ModalItem* that opens the `url` in a modal dialog instead of the sideframe, and returns it.

add_sideframe_item(*name*, *url*, *active=False*, *disabled=False*, *extra_classes=None*, *on_close=None*, *side=<object object>*, *position=None*)

Adds a *SideframeItem* that opens `url` in the sideframe and returns it.

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()*. The return value of this method is safe to use as the *position* argument of the various APIs to add items.

find_items(*item_type*, ***attributes*)

Returns a list of *ItemSearchResult* objects matching all items of `item_type` (e.g. `LinkItem`).

Items whose attributes are lazy (e.g. a `gettext_lazy name`) are matched by their *source* string, not their translation. Search for an item by the original (untranslated, usually English) string – e.g. `find_items(LinkItem, name="Save")` – so the lookup works regardless of the active language. Passing the lazy itself or a translated string is not reliable across languages.

This source matching only works for single-argument lazies such as `_("Save")`. Composite lazy values – e.g. `format_lazy("Save {}", model_name)` – have no single source string and are matched by their *resolved* (translated) value, so they cannot be looked up language-independently. Give such items a stable `identifier` and search by that instead.

get_item_count()

Returns the number of items in the menu.

class cms.toolbar.items.**BaseButton**

Bases: object

class cms.toolbar.items.**ItemSearchResult**(*item*, *index*)

Bases: object

Returned by the find APIs in *ToolbarAPIMixin*.

An ItemSearchResult will have two useful attributes:

item

The item found.

index

The index of the item (its position amongst the other items).

The ItemSearchResult itself can be cast to an integer, and supports addition and subtraction of numbers. See the *position* parameter for more details, and *Control the position of items in the toolbar* for examples.

class cms.toolbar_base.**CMSToolbar**(*request*, *toolbar*, *is_current_app*, *app_path*)

Bases: object

Parameters

The methods described below for creating/modifying toolbar items share a number of common parameters:

key

a unique identifier (typically a string)

verbose_name

the displayed text in the item

position

The position index of the new item in the list of items. May be:

1. None - appends the item to the list
2. an integer - inserts the item at that index in the list
3. an object already in the list - Inserts the item into the list immediately before the object; must be a sub-class of *BaseItem*, and must exist in the list
4. an *ItemSearchResult* - inserts the item into the list immediately before the ItemSearchResult. ItemSearchResult may be treated as an integer.

on_close:

Determines what happens after closing a frame (sideframe or modal) that has been opened by a menu item. May be:

1. None - does nothing when the sideframe closes
2. *REFRESH_PAGE* - refreshes the page when the frame closes
3. a URL - opens the URLS when the frame is closed.

disabled

Greys out the item and renders it inoperable.

active

Applies to buttons only; renders the button it its 'activated' state.

side

Either *cms.constants.LEFT* or *cms.constants.RIGHT* (both unique objects denoted above as <object object>). Decides to which side of the toolbar the item should be added.

django CMS constants used in toolbars

cms.constants.**REFRESH_PAGE**

Supplied to `on_close` arguments to refresh the current page when the frame is closed, for example:

```
from cms.constants import REFRESH_PAGE

self.toolbar.add_modal_item(
    'Modal item',
    url=modal_url,
    on_close=REFRESH_PAGE
    )
```

cms.cms_toolbars.**ADMIN_MENU_IDENTIFIER**

The *Site* menu (that usually shows the project's domain name, *example.com* by default). `ADMIN_MENU_IDENTIFIER` allows you to get hold of this object easily using *cms.toolbar.toolbar.CMSToolbar.get_menu()*.

cms.cms_toolbars.**LANGUAGE_MENU_IDENTIFIER**

The *Language* menu. `LANGUAGE_MENU_IDENTIFIER` allows you to get hold of this object easily using *cms.toolbar.toolbar.CMSToolbar.get_menu()*.

cms.cms_toolbars.**PAGE_MENU_IDENTIFIER**

The *Page* menu. `PAGE_MENU_IDENTIFIER` allows you to get hold of this object easily using *cms.toolbar.toolbar.CMSToolbar.get_menu()*.

Utility functions

Utility functions provide functionality that is regularly used within the django CMS core and are also available to third party packages.

Model admin

Action buttons

class cms.admin.utils.**ChangeListActionsMixin**

Bases: `object`

ChangeListActionsMixin is a mixin for the ModelAdmin class. It adds the ability to have action buttons and a burger menu in the admin's change list view. Unlike actions that affect multiple listed items the list action buttons only affect one item at a time.

Use `get_action_list()` to register actions and *admin_action_button()* to define the button behavior.

To activate the actions make sure `"admin_list_actions"` is in the admin classes `list_display` property.

static admin_action_button(*url: str*, *icon: str*, *title: str*, *burger_menu: bool = False*, *action: str = 'get'*, *disabled: bool = False*, *keepsideframe: bool = True*, *name: str = ''*) → str

Returns a generic button supported by the ChangeListActionsMixin.

Parameters

- **url** (*str*) – Url of the action as string, typically generated by **:func:`~cms.utils.urlutils.admin_reverse`_**
- **icon** (*str*) – Name of the icon shown in the button or before the title in the burger menu.
- **title** (*str*) – Human-readable string describing the action.

- **burger_menu** (*bool*) – If `True` the action item will be part of a burger menu right og all buttons.
- **action** (*str*) – Either `"get"` or `"post"` defining the html method used for the url. Some urls require a post method.
- **disabled** (*bool*) – If `True` the item is grayed out and cannot be selected.
- **keepsideframe** (*bool*) – If `False` the side frame (if open) will be closed before executing the action.
- **name** (*str*) – A string that will be added to the class list of the button/menu item: `cms-action-{{ name }}`

To add an action button to the change list use the following pattern in your admin class:

```
def my_custom_button(self, obj, request, disabled=False):
    # do preparations, e.g., check permissions, get url, ...
    url = admin_reverse("...", args=[obj.pk])
    if permissions_ok:
        return self.admin_action_button(
            url, "info", _("View usage"), disabled=disabled
        )
    return ""  # No button
```

get_actions_list() → list[Callable[[Model, HttpRequest], str]]

Collect list actions from implemented methods and return as list. Make sure to call it's `super()` instance when overwriting:

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

    def get_actions_list(self):
        return super().get_actions_list() + [
            self.my_first_action,
            self.my_second_action,
        ]
```

get_admin_list_actions(*request: HttpRequest*) → Callable[[Model], str]

Method to register the admin action menu with the admin's list display

Usage (in your model admin):

```
class MyModelAdmin(AdminActionsMixin, admin.ModelAdmin):
    ...
    list_display = ("name", ..., "admin_list_actions")
```

Grouper admin

class cms.admin.utils.**GrouperModelAdmin**(*model*, *admin_site*)

Bases: *ChangeListActionsMixin*, `ModelAdmin`

Easy-to-use ModelAdmin for grouper models. Usage example:

```
class MyGrouperAdmin(GrouperModelAdmin):
    # Add language tabs to change and add views
    extra_grouping_fields = ("language",)
```

(continues on next page)

(continued from previous page)

```
    # Add grouper and content fields to change list view
    # Add preview and settings action to change list view
    list_display = ("field_in_grouper_model", "content__field_in_content_model",
↪"admin_list_actions")

    # Automatically add content fields to change form (either the standard form or␣
↪any form given
    form = MyChangeForm

    ...
```

Using `GrouperModelAdmin` instead of `ModelAdmin` adds a view standard functions to your admin class to make it more easily and more consistently customizable.

1. **By adding "admin_list_actions" to the admin's `list_display`**
   attribute the change list view gets an action column as described by *`ChangeListActionsMixin`*.
2. **The admin class automatically creates a method for each field of the content model form (default: all fields)**
   named `content__{content_model_field_name}`. Those fields can be used in `list_display` just as grouper model fields. Currently, they are not sortable, however.
3. **The change form is amended with exactly those content fields also named `content__{content_model_field_name}`.**
   As a result, the change form can (but does not have to) contain both grouper model fields and content model fields. The admin takes care of creating the necessary model instances.

changeform_view(*request: HttpRequest*, *object_id: str | None = None*, *form_url: str = ''*, *extra_context: dict | None = None*) → HttpResponse

Update grouping field properties for both add and change views

delete_view(*request: HttpRequest*, *object_id: str*, *extra_context: dict | None = None*) → HttpResponse

Update grouping field properties for delete view

get_actions_list() → list

Collect list actions from implemented methods and return as list. Make sure to call it's `super()` instance when overwriting:

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

    def get_actions_list(self):
        return super().get_actions_list() + [
            self.my_first_action,
            self.my_second_action,
        ]
```

get_changelist(*request: HttpRequest*, ***kwargs*) → type

Allow for extra grouping fields as a non-filter parameter

get_changelist_instance(*request: HttpRequest*) → GrouperChangeListBase

Update grouping field properties and get changelist instance

get_content_field(*obj: Model*, *field_name: str*, *request: HttpRequest | None = None*) → Any

Retrieves the content of a field stored in the content model. If request is given extra grouping fields are processed before.

get_content_readonly_message(*request*, *content_obj*)

Return `None` if the content can be changed, otherwise a human-readable message explaining why it is read-only.

Permissions are owned by django CMS; editability (and its explanation) is owned by the content object's `is_editable` method. That method may return either a plain `bool` or a "rich bool" (e.g. from djangocms-versioning) that is falsy but also carries the reason(s) the editability checks failed via `str()`. Plain bools carry no reason, so a generic message is used.

get_extra_context(*request: HttpRequest*, *object_id: str | None = None*) → dict[str, Any]

Provide the grouping fields to the change view.

get_extra_grouping_field(*field*)

Retrieves the current value for grouping fields - by default by calling self.get_<field>, e.g., self.get_language(). If those are not implemented, this method will fail.

get_form(*request: HttpRequest*, *obj: Model | None = None*, ***kwargs*) → type

Adds the language from the request to the form class

get_grouper_obj(*obj: Model*) → Model

Get the admin object. If obj is a content object assume that the admin object resides in the field named after the admin model. The admin model name must be the same as the content model name minus "Content" at the end.

get_grouping_from_request(*request: HttpRequest*) → None

Retrieves the current grouping selectors from the request

get_language() → str

Hook on how to get the current language. By default, if it is set as a property, use the property, otherwise let Django provide it.

get_language_from_request(*request: HttpRequest*) → str

Hook for get_language_from_request which by default uses the cms utility

get_language_tuple(*site: Site | None = None*) → tuple[tuple[str, str], ...]

Hook on how to get all available languages for the language selector.

get_prepopulated_fields(*request: HttpRequest*, *obj: Model | None = None*) → dict

Drop prepopulated entries whose key is read-only.

`AdminForm` looks up every prepopulated key in the rendered form, so an entry keyed on a field that `get_readonly_fields` returns would raise `KeyError`. Filter dynamically against the class attribute so subclasses can keep declaring `prepopulated_fields` the normal way.

get_preserved_filters(*request: HttpRequest*) → str

Always preserve grouping get parameters! Also, add them to changelist filters: * Save and continue will keep the grouping parameters * Save and returning to changelist will keep the grouping parameters

get_queryset(*request: HttpRequest*) → QuerySet

Annotates content fields with the name "content__{field_name}" to the grouper queryset if for all content fields that appear in the

get_readonly_fields(*request: HttpRequest*, *obj: Model | None = None*)

Allow access to content fields to be controlled by a method "can_change_content": This allows versioned content to be protected if needed

get_search_fields(*request*)

Return search fields for either grouper model or content model

get_search_results(*request*, *queryset*, *search_term*)

Return a tuple containing a queryset to implement the search and a boolean indicating if the results may contain duplicates.

history_view(*request: HttpRequest*, *object_id: str*, *extra_context: dict | None = None*) → HttpResponse

Update grouping field properties for history view

save_model(*request: HttpRequest*, *obj: Model*, *form: Form*, *change: bool*) → None

Save/create both grouper and content object

view_on_site(*obj: Model*) → str | None

Returns True when the argument is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

content_model: Model | None = None

The content model class to be used. Defaults to the model class named like the grouper model class plus `"Content"` at the end from the same app as the grouper model class, e.g., `BlogPostContent` if the grouper is `BlogPost`.

content_related_field: str | None = None

Name of the inverse relation field giving the set of content models belonging to a grouper model. Defaults to the first field found as an inverse relation. If you have more than one inverse relation please make sure to specify this field. An example would be if the blog post content model contained a many-to-many relationship to the grouper model for, say, related blog posts.

property current_content_filters: dict[str, Any]

Filters needed to get the correct content model instance

extra_grouping_fields: tuple[str, ...] = ()

Indicates additional grouping fields such as `"language"` for example. Additional grouping fields create tabs in the change form and a dropdown menu in the change list view.

> **Note**
>
> All fields serving as extra grouping fields must be part of the admin's `fieldsets` setting for `GrouperModelAdmin` to work properly. In the change form the fields will be invisible.

grouper_field_name: str | None = None

The name of the `ForeignKey` in the content model that points to the grouper instance. If not given it is assumed to be the snake case name of the grouper model class, e.g. `"blog_post"` for the `"BlogPost"` model.

Placeholders

cms.utils.placeholder.**get_placeholder_from_slot**(*placeholder_relation: Manager*, *slot: str*, *template_obj=None*, *default_width: int | None = None*) → *Placeholder*

Retrieves the placeholder instance for a PlaceholderRelationField either by scanning the template of the template_obj (if given) or by creating or getting a Placeholder in the database

cms.utils.placeholder.**get_declared_placeholders_for_obj**(*obj: Model | EmptyPageContent | None*) → list[DeclaredPlaceholder]

Returns declared placeholders for an object. The object is supposed to either have a method `get_placeholder_slots` which returns the list of placeholders or a method `get_template` which returns

the template path as a string that renders the object. `get_declared_placeholders` returns a list of placeholders used in the template by the `{% placeholder %}` template tag.

Plugins

`cms.utils.plugins.`**`get_plugins`**(*request*, *placeholder*, *template*, *lang=None*)

Get a list of plugins for a placeholder in a specified template. Respects the placeholder's cache.

Parameters

- **`request`** – (HttpRequest) The HTTP request object.
- **`placeholder`** – (Placeholder) The placeholder object for which to retrieve plugins.
- **`template`** – (Template) The template object in which the placeholder resides (not used).
- **`lang`** – (str, optional) The language code for localization. Defaults to None.

Returns:

list: A list of plugins for the specified placeholder in the template.

Raises:

None.

Examples:

```
# Get plugins for a placeholder in a template
plugins = get_plugins(request, placeholder, template)

# Get plugins for a placeholder in a template with specific language
plugins = get_plugins(request, placeholder, template, lang="en")
```

`cms.utils.plugins.`**`assign_plugins`**(*request*, *placeholders*, *template=None*, *lang=None*)

Fetch all plugins for the given `placeholders` and cast them down to the concrete instances in one query per type.

Parameters

- **`request`** – The current request.
- **`placeholders`** – An iterable of placeholder objects.
- **`template`** – (optional) The template object.
- **`lang`** – (optional) The language code.

This method assigns plugins to the given placeholders. It retrieves the plugins from the database based on the placeholders and the language. The plugins are then downcasted to their specific plugin types.

The plugins are split up by placeholder and stored in a dictionary where the key is the placeholder ID and the value is a list of plugins.

For each placeholder, if there are plugins assigned to it, the plugins are organized as a layered tree structure. Otherwise, an empty list is assigned.

The list of all plugins for each placeholder is stored in the *_all_plugins_cache* attribute of the placeholder, while the list of root plugins is stored in the *_plugins_cache* attribute

`cms.utils.plugins.`**`has_reached_plugin_limit`**(*placeholder*, *plugin_type*, *language*, *template=None*)

Checks if the global maximum limit for plugins in a placeholder has been reached. If not then it checks if it has reached its maximum plugin_type limit.

Parameters: - placeholder: The placeholder object to check the limit for. - plugin_type: The type of plugin to check the limit for. - language: The language code for the plugins. - template: The template object for the placeholder. Optional.

Returns: - False if the limit has not been reached.

Raises: - PluginLimitReached: If the limit has been reached for the placeholder.

cms.utils.plugins.**get_plugin_class**(*plugin_type: str*) → type[*CMSPluginBase*]

Returns the plugin class for a given plugin_type (str)

cms.utils.plugins.**get_plugin_model**(*plugin_type: str*) → *CMSPlugin*

Returns the plugin model class for a given plugin_type (str)

cms.utils.plugins.**get_plugins_as_layered_tree**(*plugins*)

Given an iterable of plugins ordered by position, returns a deque of root plugins with their respective children set in the child_plugin_instances attribute.

cms.utils.plugins.**copy_plugins_to_placeholder**(*plugins*, *placeholder*, *language=None*, *root_plugin=None*, *start_positions=None*)

Copies an iterable of plugins to a placeholder

Parameters

- **plugins** (*iterable*) – Plugins to be copied
- **placeholder** (*cms.models.pluginmodel.CMSPlugin* instance) – Target placeholder
- **language** (*str*) – target language (if no root plugin is given)
- **root_plugin**
- **start_positions** (*int*) – Cache for start positions by language

The logic of this method is the following:

1. Get bound plugins for each source plugin
2. Get the parent plugin (if it exists)
3. then get a copy of the source plugin instance
4. Set the id/pk to None to it the id of the generic plugin instance above; this will effectively change the generic plugin created above into a concrete one
5. find the position in the new placeholder
6. save the concrete plugin (which creates a new plugin in the database)
7. trigger the copy relations
8. return the plugin ids

cms.utils.plugins.**downcast_plugins**(*plugins: Iterable[*CMSPlugin*]*, *placeholders: list | None = None*, *select_placeholder: bool = False*, *request: HttpRequest | None = None*) → Iterable[*CMSPlugin*]

Downcasts the given list of plugins to their respective classes. Ignores any plugins that are not available.

Parameters

- **plugins** (*List[CMSPlugin]*) – List of plugins to downcast.
- **placeholders** (*Optional[List[Placeholder]]*) – List of placeholders associated with the plugins.
- **select_placeholder** (*bool*) – If True, select_related the plugin queryset with placeholder.
- **request** (*Optional[HttpRequest]*) – The current request.

Returns

Generator that yields the downcasted plugins.

Return type

Generator[*CMSPlugin*, None, None]

cms.utils.plugins.**get_bound_plugins**(*plugins*)

Get the bound plugins by downcasting the plugins to their respective classes. Raises a KeyError if the plugin type is not available.

Creates a map of plugin types and their corresponding plugin IDs for later use in downcasting. Then, retrieves the plugin instances from the plugin model using the mapped plugin IDs. Finally, iterates over the plugins and yields the downcasted versions if they have a valid parent. Does not affect caching.

Parameters

plugins (*List[CMSPlugin]*) – List of CMSPlugin instances.

Returns
Generator that yields the downcasted plugins.
Return type
Generator[*CMSPlugin*, None, None]

Example:

```
plugins = [plugin_instance1, plugin_instance2]
for bound_plugin in get_bound_plugins(plugins):
    # Do something with the bound_plugin
    pass
```

Added 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`.

Before making a wizard available to the CMS it 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:

class cms.wizards.wizard_base.**WizardBase**(*title*, *weight*, *form*, *model=None*, *template_name=None*, *description=None*, *edit_mode_on_success=True*)

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
Whether the user will get redirected to object edit url after a successful creation or not. This only works if the object is registered for toolbar enabled models.

Important

As of version 4 of django CMS wizards are no longer registered with the `wizard_pool`. Instead you need to create a app config in `cms_config.py` to register wizards.

Added in version 4.0.

Wizards are made available to the CMS by adding a *cms.app_base.CMSAppConfig* subclass to your apps's *cms_config.py*. As an example, here's how the CMS itself registers its wizards:

```
class CMSCoreConfig(CMSAppConfig):
    cms_enabled = True  # Use cms core's functionality
    cms_wizards = [cms_page_wizard, cms_subpage_wizard]   # Namely, those wizards
```

For the above example the configuration might look like this:

```
from .cms_wizards import my_app_wizard

class MyAppConfig(CMSAppConfig):
    cms_enabled = True
    cms_wizards = [my_app_wizard]
```

Wizard class

class cms.wizards.wizard_base.**Wizard**(*title*, *weight*, *form*, *model=None*, *template_name=None*, *description=None*, *edit_mode_on_success=True*)

All wizard classes should inherit from `cms.wizards.wizard_base.Wizard`. This class implements a number of methods that may be overridden as required.

get_description(***kwargs*)

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_success_url(*obj*, ***kwargs*)

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. If the object is registered for toolbar enabled models, the object edit url will be returned. This may be modified to return the preview url instead by setting the wizard property `edit_mode_on_success` to False.

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.

Parameters

- **obj** (*object*) – The created object
- **kwargs** (*dict*) – Arbitrary keyword arguments

get_title(***kwargs*)

Simply returns the `title` property assigned during instantiation. Override this method if this needs to be determined programmatically.

get_weight(***kwargs*)

Simply returns the `weight` property assigned during instantiation. Override this method if this needs to be determined programmatically.

user_has_add_permission(*user*, ***kwargs*)

Returns boolean reflecting whether the given «user» has permission to add instances of this wizard's associated model. Can be overridden as required for more complex situations.

Parameters

user – The current user using the wizard.

Returns

True if the user should be able to use this wizard.

property id

To construct a unique ID for each wizard, we start with the module and class name for uniqueness, we hash it because a wizard's ID is displayed in the form's markup, and we'd rather not expose code paths there.

cms.wizards.wizard_base.**get_entry**(*entry_key*)

Returns a wizard object based on its *id*.

cms.wizards.wizard_base.**get_entries**()

Returns a list of Wizard objects (for all registered wizards) ordered by weight

`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 get_entries():
    # do something with a wizard...
```

cms.wizards.wizard_base.**entry_choices**(*user*, *page*, *site=None*)

Yields a list of wizard entry tuples of the form (wizard.id, wizard.title) that the current user can use based on their permission to add instances of the underlying model objects.

wizard_pool

Warning

The wizard pool is deprecated since version 4.0 and will be removed in a future version.

cms.wizards.wizard_pool.**wizard_pool**

Warning

Deprecated since version 4.0.

Using wizard_pool is deprecated. Use *cms.wizards.helper* functions instead. Since django CMS version 4 wizards are registered with the cms using *cms.app_base.CMSAppConfig* in `cms_config.py`.

class cms.wizards.wizard_pool.**WizardPool**

Deprecated since version 4.0.

get_entry(*entry*)

Deprecated since version 4.0: use `cms.wizards.helpers.get_entry()` instead

Returns the wizard from the pool identified by «entry», which may be a Wizard instance or its "id" (which is the PK of its underlying content-type).

is_registered(*entry*, ***kwargs*)

Deprecated since version 4.0.

Returns True if the provided entry is registered.

register(*entry*)

Deprecated since version 4.0.

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(*entry*)

Deprecated since version 4.0.

If «entry» is registered into the pool, remove it.

Returns True if the entry was successfully registered, else False.

Icons reusable for plugins

django CMS comes with a set of icons stored in its own icon font. The icons are based on FontAwesome4 and Bootstrap Icons.

They are available in the frontend editor (i.e. if the toolbar is available). To use them on the admin site where all the plugin editing etc. happens, you will have to load them explicitly.

```
from cms.utils.urlutils import static_with_version


class MyAdmin(admin.Admin):
    class Media:
        css = {"all": (static_with_version("cms/cms.icons.css"),)}

    ...
```

Icons are used by adding snippets like this to your templates

```
<span class="cms-icon cms-icon-<iconname>"></span>
```

The following icons are available:

- advanced-settings
- alias
- apphook
- archive
- bin
- comment
- compare

- copy
- cut
- edit
- edit-new
- highlight
- home
- info
- lock
- manage-versions
- moderate
- paste
- plugins
- publish
- redo
- rename
- search
- settings
- sitemap
- undo
- unlock
- unpublish
- view

Example:

```
<span class="cms-icon cms-icon-edit-new me-2"></span>{% translate "Edit new..." %}
```

5.3.5 Release notes & upgrade information

Some versions of django CMS present more complex upgrade paths than others, and some **require** you to take action. It is strongly recommended to read the release notes carefully when upgrading.

It goes without saying that you should **backup your database** before embarking on any process that makes changes to your database.

5.1.0a1 release notes

February 14, 2026

Welcome to django CMS 5.1.0a1!

These release notes cover the new features, as well as some backwards incompatible changes you'll want to be aware of when upgrading from django CMS 4.0 or earlier. We've begun the deprecation process for some features.

See the How to upgrade to 5.1.0a1 to a newer version guide if you're updating an existing project.

Django and Python compatibility

django CMS supports **Django 4.2, 5.0, 5.1, and 5.2**. We highly recommend and only support the latest release of each series.

It supports **Python 3.11, 3.12, 3.13, and 3.14**. As for Django we highly recommend and only support the latest release of each series.

How to upgrade to 5.1.0

What's new in 5.1.0

New UI design

The most prominent change in django CMS 5.1.0 is the new user interface design. While functionally similar to the previous design, it offers a more modern and streamlined experience, better color contrast, and more consistency between the frontend editor and Django's admin interface.

Key changes include:

- More Django-like green based color prefers-color-scheme
- Modern design with round corners and updated icons
- Improved visual hierarchy and spacing

It is active by default, but the legacy design can be re-enabled by adding the `data-cms-theme` attribute with the value `4` to the `<html>` tag in your templates.

More flexible site configurations

The Django admin app does not have to be available on all sites. You can now configure which sites have access to the admin interface and still edit all sites using the admin's frontend editing and preview endpoints. This is especially useful if you do not want to expose the admin interface on external production sites.

You can now run multiple sites with one settings file. If you omit the `SITE_ID` in your settings, django CMS will determine the current site using Django's site framework based on the request's host header. This allows you to run multiple sites with one instance and one settings file. Note, that apphooks must not share the same URLs in such a scenario to avoid conflicts.

If you need more complex site determination logic, you can implement a custom middleware that sets `request.site` accordingly. It will be respected by django CMS.

For a step-by-step guide and examples, see *Multi-Site Installation* (including notes on apphook isolation and URL conflicts). For management commands that run without an HTTP request, consult *Command Line Interface* for options like `--site` to select the target site context.

Model-specific plugins

Plugins can now be restricted to specific models using two complementary filters:

- Plugin-level: `CMSPluginBase.allowed_models` — list of model identifiers (`"app_label.modelname"`) with the two special values `None` (allowed everywhere) and `[]` (allowed nowhere).
- Model-level: `Model.allowed_plugins` — list of plugin class names, with the special value `None` (all plugins allowed, subject to their own `allowed_models`).

Both filters must pass for the frontend editor offer plugins to the users. This is particularly useful for plugins that are bound to specific Django models and should not be available to editors on other models - or to restrict certain models to a curated set of plugins.

Example:

```
class TextFieldPlugin(CMSPluginBase):
    render_template = "forms/fields/text.html"
    allowed_models = ["my_form_app.form"]

class Form(models.Model):
    fields = PlaceholderRelationField("fields")
    allowed_plugins = ["TextFieldPlugin"]
```

See also the how-to guide for details and more examples: *Restricting plugins to specific models*.

Simplified project creation with the `djangocms` command

Setting up django CMS is now a one-liner. The new `djangocms` command (also available as `python -m cms`) scaffolds a complete, ready-to-run project, installs its requirements, runs the migrations, creates a superuser and checks the installation for you:

```
djangocms myproject
```

Run without a project name it drops into an **interactive mode** that asks for the name and walks you through every option. You can also drive it entirely from the command line, for example:

```
djangocms myproject --mode headless --no-versioning --stories
```

The most important options are:

- `--mode {traditional,headless,hybrid}` — choose how the CMS serves content (default: `traditional`).
- `--versioning` / `--no-versioning` — add content versioning with djangocms-versioning (default: on).
- `--moderation` / `--no-moderation` — add content moderation (builds on versioning; default: off).
- `--alias` / `--no-alias` — add reusable aliases (default: on).
- `--stories` / `--no-stories` — add the stories component library (default: off).
- `--interactive` / `--noinput` — force or suppress all prompts.

Adding django CMS to an existing project

The same command can now **add django CMS to an existing Django project** instead of scaffolding a new one. Run it from your project root with a project name of `.`:

```
djangocms .
```

django CMS then reads your project's settings module from `manage.py` and makes best-effort, automated edits to wire everything up:

- updates `INSTALLED_APPS`, `MIDDLEWARE` and the template context processors,
- creates a project `templates` directory with a base template if your project does not have one yet,
- appends the required django CMS settings,
- adds the CMS URL patterns to your `ROOT_URLCONF` (respecting `i18n_patterns` when already in use), and
- installs the packages needed for the selected options, then runs the migrations and `cms check`.

The same `--mode` and component options above apply, so you can tailor the installation to your needs. The edits are automated and should be reviewed afterwards, so make sure your project is under version control or backed up before running it. The command is idempotent: running it again will not duplicate the changes.

You can preview everything first with `--dry-run`, which prints a unified diff of the changes it would make — without writing any files or installing anything:

```
djangocms . --dry-run
```

The rules that govern these edits are fetched from the `cms-template` repository (matching your installed django CMS version), with a bundled copy serving as an offline fallback, so the integration can keep up with the ecosystem without a new django CMS release.

See *Installing django CMS* for a step-by-step walkthrough.

Minor features

- Plugin parent/child restrictions are more expressive. The `child_classes` and `parent_classes` attributes (and their `CMS_PLACEHOLDER_CONF` counterparts) now accept glob patterns such as `"Bootstrap*"` or `"*Link*"`, which are expanded against the names of all registered plugins. An empty list (`[]`) now consistently means "none allowed" (no children, or no parent — i.e. placeholder only), whereas `None` continues to mean "no restriction". As a special case, `parent_classes = ["*"]` requires the plugin to have a parent (any plugin). Expansion is cached, so restriction evaluation stays fast on the rendering path. In addition, `child_classes = "auto"` allows exactly those plugins that explicitly name this plugin in their `parent_classes`, so children opt in to a parent rather than the parent having to list every allowed child.
- Placeholder objects now have a *`delete_plugins()`* method which efficiently bulk-deletes multiple plugins from the placeholder. It takes an iterable of plugin objects as a parameter.
- Async support: User middleware and apphook registration support async setups and now let you run django CMS using asgi. Database access is avoided when registering apphook URLs to make them async-safe.
- Apphooks now by default allow access to the page content object's placeholders on the apphook's root page.
- Allow page permission change from advanced settings with appropriate permissions
- Add icon for redirected pages in page tree
- Let the "eye" icon in the page tree directly go to edit endpoints for editable content
- Preserve GET parameters when switching to preview or edit mode
- Improved UX for external placeholders (e.g., static aliases)
- Optimized placeholder and plugin utilities for better performance
- Re-introduced help menu
- Django 6 compatibility support
- Introduce contracts for django CMS Extensions
- Clearer page permission admin: the `can_view` field is now labelled "can view restricted pages" with help text explaining that granting view access also restricts the page, the `can_change_permissions` help text spells out what it controls, and the "edit permissions required" validation messages now name the exact "Can edit" checkbox to enable.

Bug Fixes

- Get_permissions failed for missing global permissions
- Create page wizard not available on empty install
- Make Placeholder.add_plugin() set plugin.instance to self
- Resolve empty page_title in frontend edit mode for apphooks
- Improve anti-aliasing of django CMS logo png
- Use PageContent.template_choices attribute for template choices instead of settings
- Existing child plugin restriction cannot be reduced to an empty list (which implies all plugins allowed)
- Gracefully handle unresolvable extends variables in placeholder scanning
- Allow frontend-editable models to omit get_template method
- Safe fallback for includes when scanning for placeholders
- Fix ApphookReloadMiddleware not handling new language variants
- Copying failed if a target placeholders was missing
- Grouper admin kept read-only fields as prepopulated fields

Backward incompatible changes in 5.1.0

Features deprecated in 5.1.0

Page model

- The manager method `Page.objects.get_title(page, language, language_fallback=False)` will be removed in django CMS 6.0. Use `page.get_content_obj()` or `page.get_admin_content()` instead.
- `Page` attribute `page.parent_page` is deprecated. Instead use the attribute `page.parent`.
- `Page` method `page.get_parent_page()` is deprecated and will be removed in django CMS 6.0. Instead use the *page.parent* attribute.
- `Page` attribute `page.languages` is deprecated and will be removed in django CMS 6.0. Use `page.get_languages()` instead.
- `Page` methods `page.remove_language()` and `page.update_languages()` are deprecated and will be removed in django CMS 6.0. They have no effect any more.
- `Page` method `page.get_published_languages()` is deprecated and will be removed in django CMS 6.0. Use `page.get_languages(admin_manager=False)` instead.
- `Page` method `page.set_translations_cache()` is deprecated and will be removed in django CMS 6.0. Use `page.get_content_obj()` instead - it leaves the translations cache populated. For admin views use `page.set_admin_content_cache()` instead.
- `Page` methods `page.copy()` and `page.copy_descendant()``have a keyword argument ``parent_node` which has been renamed to `parent_page`. `parent_node` is deprecated and will be removed in django CMS 6.0.

Page permissions

- `PagePermission``method ``permission.get_page_ids()` is deprecated and will be removed in django CMS 6.0. Use `permission.get_page_permission_tuple()` instead for faster permission checks by path and not id.

Plugin rendering

- `BaseRenderer` method `renderer.get_placeholder_toolbar_js()` has a keyword argument `page` which will be removed in django CMS 6.0. It can already be safely removed from all calls.
- The `parents` keyword argument of `CMSPlugin.get_plugin_info()`, `cms.toolbar.utils.get_plugin_toolbar_info()` and `cms.toolbar.utils.get_plugin_toolbar_js()` is deprecated and will be removed in django CMS 6.0. Parent plugin restrictions are no longer sent to the frontend – droppability is derived from the child restrictions of each potential parent (and of the placeholder) – so the argument and the `plugin_parent_restriction` entry it produced can be safely removed from all calls.

Utility functions

- Most `cms.utils.i18n` utility functions accepting an optional `site_id` argument will require this argument in django CMS 6.0. These functions are `get_languages()`, `get_language_code()`, `get_language_list()`, `get_language_tuple()`, `get_language_dict()`, `get_public_languages()`, `get_language_object()`, `get_language_objects()`, `get_default_language()`, `get_fallback_languages()`, `get_redirect_on_fallback()`, and `hide_untranslated()`.
- `cms.utils.page_permissions.user_can_add_page()` and `cms.utils.page_permissions.has_generic_permission` accepting an optional `site` argument will require this argument in django CMS 6.0.
- `cms.utils.page.get_page_queryset()` is deprecated and will be removed in django CMS 6.0. Use `Page.objects.on_site(site)` instead.
- `cms.utils.placeholder.get_toolbar_plugin_struct()` has a keyword argument `page` which has been renamed to `obj`. `page` is deprecated and will be removed in django CMS 6.0.

Wizard helpers

- The `cms.wizards.helpers` module has been deprecated and will be removed in django CMS 6.0. Use `cms.wizards.wizard_base.get_entries()` and `cms.wizards.wizard_pool.get_entry()` instead.

Removal of deprecated functionality

API

`cms.api.create_title` has been removed. Use *`cms.api.create_page_content()`* instead.

`StaticPlaceholder`

- `StaticPlaceholder` has been removed from `cms.models` and `cms.admin`. Use static aliases instead.
- This implies that `cms.plugin_rendering` does not accept static placeholders any more.
- The `static_placeholder` template tag has been removed.

Placeholders

- `PlaceholderField` has been removed from `cms.fields`. Use *`PlaceholderRelationField`* in conjunction with *`get_placeholder_from_slot()`* instead (also see *How to use placeholders outside the CMS*).
- `PlaceholderAdminMixin` has been removed from `cms.admin`. It is not needed anymore and can be safely removed from your code.
- A placeholder's `actions` property (deprecated in django CMS 5) has been removed. This also includes the removal of the two classes `cms.utils.placeholder.PlaceholderNoAction` and `cms.utils.placeholder.MLNGPlaceholderActions`. If you need these classes, move them to your own codebase.

Permissions

`cms.utils.permissions.has_page_permission()` has been removed. Use `cms.utils.page_permissions.has_generic_permission()` instead.

As part of performance improvements, the following methods have been removed from `cms.utils.page_permissions`. They were deprecated in django CMS 4.1:

- `cms.utils.page_permissions.get_add_ids()`
- `cms.utils.page_permissions.get_change_ids()`
- `cms.utils.page_permissions.get_change_advanced_settings_ids()`
- `cms.utils.page_permissions.get_change_permissions_ids()`
- `cms.utils.page_permissions.get_get_delete_ids()`
- `cms.utils.page_permissions.get_move_page_ids()`
- `cms.utils.page_permissions.get_publish_ids()`
- `cms.utils.page_permissions.get_view_ids()`
- `cms.utils.permissions.get_view_restrictions()`

Menus

- `cms.cms_menus.get_visible_nodes` has been removed. Use `cms.cms_menus.get_visible_page_contents` instead.
- `cms.cms_menus.get_menu_node_for_page` has been removed. Use `cms.cms_menus.get_menu_node_for_page_content` instead.

Miscellaneous

- The `title` property has been removed from `cms.cms_toolbars.PageToolbar`. Use `cms.cms_toolbars.PageTooolbar.page_content` instead.
- `cms.forms.get_root_nodes` has been removed. Use `cms.cms_menus.get_root_pages` instead.
- `cms.extensions.toolbar.ExtensionToolbar.get_title_extension_admin` has been removed. Use `cms.extensions.toolbar.ExtensionToolbar.get_page_content_extension_admin` instead.
- `cms.toolbar.utils.get_plugin_tree_as_json` has been removed. Use `cms.toolbar.utils.get_plugin_tree` and convert the result to JSON instead.

5.0.5 release notes

November 12, 2025

Welcome to django CMS 5.0.5!

These release notes cover the new features, as well as some backwards incompatible changes you'll want to be aware of when upgrading from django CMS 4.1 or earlier. We've begun the deprecation process for some features.

Django and Python compatibility

django CMS supports **Django 4.2, 5.0, 5.1, 5.2, and 6.0**. We highly recommend and only support the latest release of each series.

It supports **Python 3.10, 3.11, 3.12, 3.13**. As for Django we highly recommend and only support the latest release of each series.

How to upgrade to 5.0.5

Update your project's `requirements.txt` file to require (at least) django CMS 5.0.5 and run `pip install -r requirements.txt`.

If you are upgrading from an earlier version of django CMS, read the release notes for all the versions between your current version and this one. Check the *release notes* for each version to see if there are any special instructions.

Run migrations:

```
python -m manage migrate
```

Warning

Since the migrations merge the `TreeNode` and `Page` models, you should run the migration in a test environment first to ensure that the migration runs successfully.

What's new in 5.0.5

Bug Fixes:

- Pin language of toolbar update to the request language
- Ensure edit endpoint language selection when admin is not using i18n_patterns (#8367) (#8390) (9e00f694e) – Fabian Braun
- Copying x-language lead to unique constraint violation (#8366) (#8386)
- Avoid escaping (= stringify) None-values in PageAttribute-TemplateTag (#8375) (#8384)
- Set default value for edit_fields parameter to avoid AttributeError (#8381)
- Link syntax in welcome.html
- Searching pages for language-specific content failed due to wrong search queryset (#8355) (#8358)

Statistics:

This release includes 5 pull requests, and was created with the help of the following contributors (in alphabetical order):

- Fabian Braun (7 pull requests)

With the review help of the following contributors:

- Vinit Kumar

Thanks to all contributors for their efforts!

5.0.4 release notes

October 3, 2025

Welcome to django CMS 5.0.4!

These release notes cover the new features, as well as some backwards incompatible changes you'll want to be aware of when upgrading from django CMS 4.1 or earlier. We've begun the deprecation process for some features.

Django and Python compatibility

django CMS supports **Django 4.2, 5.0, 5.1, and 5.2**. We highly recommend and only support the latest release of each series.

It supports **Python 3.10, 3.11, 3.12, and 3.13**. As for Django we highly recommend and only support the latest release of each series.

How to upgrade to 5.0.4

Update your project's `requirements.txt` file to require (at least) django CMS 5.0.3 and run `pip install -r requirements.txt`.

If you are upgrading from an earlier version of django CMS, read the release notes for all the versions between your current version and this one. Check the *release notes* for each version to see if there are any special instructions.

Run migrations:

```
python -m manage migrate
```

Warning

Since the migrations merge the `TreeNode` and `Page` models, you should run the migration in a test environment first to ensure that the migration runs successfully.

What's new in 5.0.4

Bug Fixes:

- Wrong placeholders rendered when using apphooks with own placeholders (#8343) (#8348) – Fabian Braun

Statistics:

This release includes 1 pull request, and was created with the help of the following contributors (in alphabetical order):

- Fabian Braun (1 pull request)

With the review help of the following contributors:

- Vinit Kumar

Thanks to all contributors for their efforts!

5.0.3 release notes

September 17, 2025

Welcome to django CMS 5.0.3!

These release notes cover the new features, as well as some backwards incompatible changes you'll want to be aware of when upgrading from django CMS 4.1 or earlier. We've begun the deprecation process for some features.

Django and Python compatibility

django CMS supports **Django 4.2, 5.0, 5.1, and 5.2**. As for Django, we highly recommend and only support the latest release of each series.

It supports **Python 3.10, 3.11, 3.12, and 3.13**. As for Django we highly recommend and only support the latest release of each series.

How to upgrade to 5.0.3

Update your project's `requirements.txt` file to require (at least) django CMS 5.0.3 and run `pip install -r requirements.txt`.

If you are upgrading from an earlier version of django CMS, read the release notes for all the versions between your current version and this one. Check the *release notes* for each version to see if there are any special instructions.

Run migrations:

```
python -m manage migrate
```

Warning

Since the migrations merge the `TreeNode` and `Page` models, you should run the migration in a test environment first to ensure that the migration runs successfully.

What's new in 5.0.3

Bug Fixes:

- Django 6 compatibility (July 2025) (8302) – Fabian Braun
- Respect individual placeholder checks if they can be changed (#8318) – Fabian Braun
- Cut children from inactive menu nodes when level is less or equal to 0 (#8324) – Stefan Wehrmeyer
- Copy lang management command - include PageUrl (#8335) – Vašek Chalupníček
- Optimize placeholder and plugin utilities (#8337) – Fabian Braun
- Migration 0033 failed when empty placeholder objects were not present in the db (#8339) – Fabian Braun

Statistics:

This release includes 9 pull requests, and was created with the help of the following contributors (in alphabetical order):

- Fabian Braun (5 pull requests)
- Stefan Wehrmeyer (1 pull request)
- Vašek Chalupníček (1 pull request)

- Github Release Action (3 pull requests)

With the review help of the following contributors:

- Fabian Braun
- Vinit Kumar

Thanks to all contributors for their efforts!

5.0.2 release notes

July 27, 2025

Welcome to django CMS 5.0.2!

These release notes cover the new features, as well as some backwards incompatible changes you'll want to be aware of when upgrading from django CMS 4.1 or earlier. We've begun the deprecation process for some features.

Django and Python compatibility

django CMS supports **Django 4.2, 5.0, 5.1, and 5.2**. As for Django, we highly recommend and only support the latest release of each series.

It supports **Python 3.10, 3.11, 3.12, and 3.13**. As for Django we highly recommend and only support the latest release of each series.

How to upgrade to 5.0.2

Update your project's `requirements.txt` file to require (at least) django CMS 5.0.2 and run `pip install -r requirements.txt`.

If you are upgrading from an earlier version of django CMS, read the release notes for all the versions between your current version and this one. Check the *release notes* for each version to see if there are any special instructions.

Run migrations:

```
python -m manage migrate
```

Warning

Since the migrations merge the `TreeNode` and `Page` models, you should run the migration in a test environment first to ensure that the migration runs successfully.

What's new in 5.0.2

Bug Fixes:

- The headless template included inline CSS (#8291) (#8292) (14dd89faf) – Fabian Braun
- GrouperAdmin Search in content model (#8284) (#8290) (46f0e0775) – Muhammad Hassan Siddiqi
- Allow clicks to propagate to plugins (#8288) (#8289) (2c1ab2d7d) – Fabian Braun
- show_placeholder did not respect edit/preview mode and failed loudly (#8272) (#8274) (e7be7bc9b) – Fabian Braun
- Allow lazy wizard initialization (#8265) (365185848) – Fabian Braun
- Apphook widgets were not always detected correctly (#8263) (ec2f4e815) – Fabian Braun

- Migrating from django CMS 3.x failed in some rare cases (#8249) (bcf7f895a) – jmit-modern
- Added default_auto_field to cms/apps.py (#8254) (4a12dd38a) – Fabian Braun
- 404 not raised in CMS view when no URLs matched (#8240) (3d069cbe5) – Stefan Wehrmeyer
- Replaced outdated usage of page__node__path with page__path (#8238) (a9105a738) – Stefan Wehrmeyer

Statistics:

This release includes 13 pull requests, and was created with the help of the following contributors (in alphabetical order):

- Fabian Braun (6 pull request)
- Github Release Action (3 pull requests)
- jmit-modern (1 pull request)
- Muhammad Hassan Siddiqi (1 pull request)
- Stefan Wehrmeyer (2 pull requests)

With the review help of the following contributors:

- Fabian Braun
- Vinit Kumar

Thanks to all contributors for their efforts!

5.0.1 release notes

May 21, 2025

Welcome to django CMS 5.0.1!

These release notes cover the new features, as well as some backwards incompatible changes you'll want to be aware of when upgrading from django CMS 4.1 or earlier. We've begun the deprecation process for some features.

Django and Python compatibility

django CMS supports **Django 4.2, 5.0, 5.1, and 5.2**. We highly recommend and only support the latest release of each series.

It supports **Python 3.10, 3.11, 3.12, and 3.13**. As for Django, we highly recommend and only support the latest release of each series.

How to upgrade to 5.0.1

Update your project's `requirements.txt` file to require (at least) django CMS 5.0.1 and run `pip install -r requirements.txt`.

If you are upgrading from an earlier version of django CMS, read the release notes for all the versions between your current version and this one. Check the *release notes* for each version to see if there are any special instructions.

Run migrations:

```
python -m manage migrate
```

> **Warning**
>
> Since the migrations merge the `TreeNode` and `Page` models, you should run the migration in a test environment first to ensure that the migration runs successfully.

What's new in 5.0.1

Bug Fixes:

- Adjust checks for GrouperAdmin to allow for *prepopulated_fields* (d6be4747e) – Fabian Braun
- Show all text-enabled plugins inside djangocms-text (0a2ce64c4) – Fabian Braun
- Structure board update sometimes failed to add all interactive elements (d040cee) – Fabian Braun
- Remove circular import in `cms.forms.validators` (1548fba) – Fabian Braun

Statistics:

This release includes 4 pull requests, and was created with the help of the following contributors (in alphabetical order):

- Fabian Braun (4 pull request)

With the review help of the following contributors:

- Jacob Rief
- Vinit Kumar

Thanks to all contributors for their efforts!

5.0.0 release notes

May 12, 2025

Welcome to django CMS 5.0.0!

These release notes cover the new features, as well as some backwards incompatible changes you'll want to be aware of when upgrading from django CMS 4.1 or earlier. We've begun the deprecation process for some features.

Django and Python compatibility

django CMS supports **Django 4.2, 5.0, 5.1, and 5.2**. We highly recommend and only support the latest release of each series.

It supports **Python 3.10, 3.11, 3.12, and 3.13**. As for Django we highly recommend and only support the latest release of each series.

How to upgrade to 5.0.0

Update your project's `requirements.txt` file to require (at least) django CMS 5.0.0 and run `pip install -r requirements.txt`.

If you are upgrading from an earlier version of django CMS, read the release notes for all the versions between your current version and this one. Check the *release notes* for each version to see if there are any special instructions.

Run migrations:

```
python -m manage migrate
```

> **Warning**
>
> Since the migrations merge the `TreeNode` and `Page` models, you should run the migration in a test environment first to ensure that the migration runs successfully.

What's new in 5.0.0

Editor experience: Improved turn-around times

Significant improvements have been made to enhance the editor experience by reducing turn-around times.

Faster plugin addition

In django CMS 5.0.0, adding child plugins has been made easier when only one plugin type is allowed. When adding a plugin to a placeholder, there is no need for a plugin selector modal if only one plugin is available. Now in this situation the plugin selector modal is skipped, streamlining the process. Typical use cases include "nested" plugins such as accordions, rows with columns, and similar structures.

Additionally, if the plugin form contains no fields to edit or has default values for all fields, a plugin can be created without add plugin form. To enable this quick creation of content, set the plugin's property *show_add_form* to `False`.

Improved plugin updates

One of the key changes is the introduction of a new mechanism for handling plugin data updates. This mechanism reduces the number request-response cycles and optimizes the rendering process, resulting in faster load times and more responsive interactions when managing content. Editors will benefit from a more efficient and enjoyable editing workflow, with quicker updates by a factor of roughly two and smoother transitions between editing tasks.

Full content updates are only necessary for so-called "non-local" changes. Plugins that change content at other places than its position on the page or its children, can be marked by setting the new *is_local* attribute to `False`. This will trigger a full content update if the plugin is changed. An example of a non-local change is a plugin that adds a title to a table of contents. If changed, both the title itself and the table of contents need to be updated.

Less server load

Global caching of plugin restrictions improves the server response time by ~100ms after changing a plugin.

Plugin restrictions need to be recalculated on every request in edit and structure mode, which is of n^2 complexity (n: number of plugins). While the plugin restrictions can be widely customized, the standard implementation respects the *CMS_PLACEHOLDER_CONF* setting, allowing restrictions to depend on template and slot names. This release caches these restrictions globally in the plugin pool for all plugins that follow the standard implementation, i.e., plugins that do not override the `get_require_parent()`, `get_parent_classes()`, `get_child_class_overrides()`, or `get_child_classes()` methods.

Performance: Optimized database access

To enhance database performance, one of the key changes is the **merging of the `TreeNode` model into the `Page` model**. This change simplifies database accesses and improves performance by reducing the number of joins required for common queries. As a result, page-related operations are faster and more efficient. Compatibility shims have been added to ensure that custom code accessing the *Page.node* attribute continues to work and will reference back to the page itself.

Additionally, this release optimizes the number of queries needed for the **edit and structure endpoints** to improve responsiveness and the editor experience. When rendering plugins, django CMS first needs to get the plugin tree (`CMSPlugin` instances) for each placeholder, and then turn each of the plugins into its actual class, such as `Text` or

`Picture`, by getting the plugin models from the database. This process is called downcasting. When downcasting plugins are now queried by their concrete model, benefiting plugins that use proxy models, such as `djangocms-frontend` or `djangocms-cascade`. Downcasting is fully avoided for plugins without a model. During downcasting, the cache for parent references is prepopulated with downcasted parent plugins, avoiding database hits when plugin templates access `plugin.parent`. Note that after downcasting `plugin.parent` now returns a downcasted plugin, not a `CMSPlugin` instance.

Plugin renderers now cache their page's template name, necessary to respect the *CMS_PLACEHOLDER_CONF* setting, solving an n+1 issue that led to a page object being fetched for each plugin type used. The `Placeholder` model's internal code has been simplified to reduce the number of database hits. Its manager's `get_for_obj()` method now automatically prepopulates the cache for the source field of fetched placeholders, solving another n+1 issue when accessing `placeholder.source`.

Security: Improved content security policy support

Earlier versions of django CMS added inline JavaScript to the page in edit mode to communicate with the frontend editor. This effectively barred projects from enforcing meaningful content security policies. In django CMS 5.0.0, we have removed all inline JavaScript from the edit mode (or other places in django CMS core), replacing it with text/json objects to communicate with the frontend editor. This allows projects to enforce strict Content Security Policies (CSP) without any issues.

For a fully working project, it is also important that other packages used, especially plugins, do not rely on inline JavaScript. This change enhances the security posture of your django CMS projects by enabling the use of CSP headers to mitigate cross-site scripting (XSS) and other code injection attacks.

Use cases: Full Headless support

Django CMS 5.0.0 is headless-ready, allowing you to use django CMS as a backend service to provide content to the frontend technology of your choice. Traditionally, django CMS serves the content as HTML pages. In headless mode, django CMS does not publish the HTML page tree. Instead, you can retrieve content via an API, such as djangocms-rest.

To run django CMS in headless mode, remove the catch-all URL pattern from your project's *urls.py* file and replace it with an API endpoint. This allows django CMS to be fully accessible through the admin interface while serving content exclusively through the API.

Additionally, you can continue running a hybrid mode where both HTML pages and API content are served.

See *How to run django CMS in headless mode*.

Development: Exception handling

Since django CMS 4, exceptions that happen during plugin rendering have been caught and displayed a message at the plugin's position. After feedback from the community, django CMS 5.0 refactored exception handling.

- Exceptions are now caught on placeholder level.
- In edit mode, a message about the exception is shown for the placeholder. If `settings.DEBUG == True` this message includes the full Django trace.
- Editors still can edit plugins causing the exception. It can be edited by double-clicking the error message or through the structure board.
- In preview mode and on site, the placeholder containing the plugin will render empty.
- If *CMS_CATCH_PLUGIN_500_EXCEPTION* is set to `False`, trying to view content that raises an exception will trigger a server error (http 500). Preview and edit modes will still work.

Minor features

- Deleting pages or deleting translations now gives a much clearer delete confirmation message. It does not list all objects deleted but summarizes how many pages, translations (counted by `PageUrl` objects) and plugins are about to be deleted.
- `CMS_PLACEHOLDER_CONF` now allows to add configuration by template name for placeholders that not necessarily are part of a page, but could be part of any model (e.g., an alias). Instead for looking at pages, the placeholder tries to access a `get_template()` method on its source model instance to identify the template name its rendered on.
- Due to the faster way of updating content in edit mode, `<script>` tags do not need to be marked with classes like `cms-trigger-event-document-DOMContentLoaded`, `cms-trigger-event-window-DOMContentLoaded`, or `cms-trigger-event-window-load`. Each script in the Sekizai `js` block is now executed in the order they are defined in the template. After all scripts have been loaded and executed, the document's `DOMContentLoaded`, the window's `load` events are triggered. The `cms-content-refresh` jQuery event on the window is also triggered for backwards compatibility.
- Plugins now inherit the `FrontendEditableMixin`. While plugins always have been frontend editable, this allows for defining parts of the rendered plugin to just edit, say, a subset of fields. Other packages, such as djangocms-text, can use this common endpoint to provide inline editing for selected fields of their plugins.

Backward incompatible changes in 5.0

Merging of Page.node into Page

To improve performance and simplify database accesses, the `TreeNode` model has been merged into the `Page` model. This change is backward incompatible and will require a database migration.

Compatibility shims have been added to the `Page` model to ensure that custom code that accesses the `Page.node` attribute will continue to work. However, this compatibility shim will be removed in django CMS 6.0 release. For now, they raise a `RemovedInDjangoCMS60Warning` warning.

Most prominent changes to custom code are:

- Pages have a `site` field again: `page.node.site` becomes `page.site`
- `page.node.path` becomes `page.path`
- `page.node.depth` becomes `page.depth`
- `page.node.numchild` becomes `page.numchild`
- `page.node.parent` and `page.page_parent` become `page.parent`

Please also check your `.filter()`, `.order()`, `.select_related()`, and `.prefetch_related()` calls to ensure they are still correct: `.filter(node__site=site)` becomes `.filter(site=site)` etc.

If you have custom code that accesses the `Page.node` attribute, you should update it to use the new attributes on the `Page` model.

Miscellaneous

- The function `cms.cms_menus.get_visible_nodes` has been deprecated. For performance reasons, the `cms_menus` builds the navigation node list based on page content objects. Use `cms.cms_menus.get_visible_page_contents` instead.
- The `cms.test_utils.testcases.CMSTestCase` class's `assertWarns` has been removed since it was an alias of `CMSTestCase.failUnlessWarns` and shadows Python's `assertWarns`. In your test cases, use Python's `assertWarns` instead, or use the `failUnlessWarns` method of `CMSTestCase` which retains the syntax of the original method.

- `CMSPluginBase.get_require_parent()`, `CMSPluginBase.get_child_class_overrides()`, `CMSPluginBase.get_child_plugin_candidates()`, `CMSPluginBase.get_child_classes()`, `CMSPluginBase.get_parent_classes()` by default receive `None` for their `page` argument.

Features deprecated in 5.0

- Use of the `node` property of the *Page* model is deprecated. Use its attributes on the *Page* model directly instead.

Removal of deprecated functionality

- Built-in alias plugin: The alias plugin has been removed. If you need this functionality, you can use the `djangocms-alias` package.
- `SuperLazyIterator`: This class has been removed. If you need this functionality, you can use the `django.utils.functional.lazy`.
- `LazyChoiceField`: This class has been removed. If you need this functionality, you can use the default `django.forms.fields.ChoiceField` class.
- `SlugWidget`: This class has been removed from `cms.wizard.forms`. If you need this functionality, you can use the `cms.admin.forms.SlugWidget` class.
- `CMSPlugin.get_breadcrumb` and `CMSPlugin.get_breadcrumb_json`: These methods have been removed. They are unused, undocumented, and were broken since django CMS 4.0.

4.1.9 release notes

September 30, 2025

> **Warning**
>
> Upgrading from previous versions
>
> django CMS 4.1 is the **first community release** of django CMS 4. Django CMS 4 introduces changes that **require** action if you are upgrading from a 3.x version. Please read the step-by-step guide to the process of upgrading from 3.5+ to 4 here: *4.0.0 release notes*

Welcome to django CMS 4.1.9!

Django and Python compatibility

django CMS supports **Django 3.2 to 5.2**. We highly recommend and only support the latest release of each series.

It supports **Python 3.9, 3.10, 3.11, and 3.12**. As for Django we highly recommend and only support the latest release of each series.

What's new in 4.1.8

Bug Fixes:

- Wrong placeholders rendered when using apphooks with own placeholder (4f68ba794) – Fabian Braun
- copy lang management command - include PageUrl (#7548) – Fabian Braun
- Migration 0033 failed when empty placeholder objects were not present in the db (#8340) (c7c0b7a35) – Fabian Braun
- Allow lazy wizard initialization (#8266) (#8283) – Fabian Braun

Statistics:

This release includes 6 pull requests, and was created with the help of the following contributors (in alphabetical order):

- Fabian Braun (3 pull request)
- Github Release Action (3 pull requests)

With the review help of the following contributors:

- Vinit Kumar

Thanks to all contributors for their efforts!

4.1.8 release notes

September 25, 2025

> **Warning**
>
> Upgrading from previous versions
>
> It is recommended to upgrade directly to django CMS 4.1.9, as version 4.1.8 contains incorrect version information and is missing important fixes.

4.1.7 release notes

April 22, 2025

> **Warning**
>
> Upgrading from previous versions
>
> django CMS 4.1 is the **first community release** of django CMS 4. Django CMS 4 introduces changes that **require** action if you are upgrading from a 3.x version. Please read the step-by-step guide to the process of upgrading from 3.5+ to 4 here: *4.0.0 release notes*

Welcome to django CMS 4.1.7!

Django and Python compatibility

django CMS supports **Django 3.2 to 5.2**. We highly recommend and only support the latest release of each series.

It supports **Python 3.9, 3.10, 3.11, and 3.12**. As for Django, we highly recommend and only support the latest release of each series.

What's new in 4.1.7

Bug Fixes:

- show_placeholder did not respect edit/preview mode and failed loudly (#8273) (c062968b7) – Fabian Braun
- Updated deprecation warning to current version scheme (#8269) (ce2dba0) – Fabian Braun
- Allow lazy wizard initialization (#8268) (04003bb) – Fabian Braun
- Migration failed in rare cases (#8249) (b5b7f30) – Fabian Braun
- Added default_auto_field to cms/apps.py (#8254) (14425d) – Fabian Braun

- Adjusted checks for GrouperAdmin to allow for *prepopulated_fields* (#8255) (7f08375) – Fabian Braun
- Remove text decorations in page tree introduced by django 5.2 (#8214) (56b75e1) – Fabian Braun
- Allow lazy wizard initialization (#8283) (e1e8f32) – Fabian Braun

Statistics:

This release includes 11 pull requests, and was created with the help of the following contributors (in alphabetical order):

- Fabian Braun (8 pull request)
- Github Release Action (3 pull requests)

With the review help of the following contributors:

- Vinit Kumar

Thanks to all contributors for their efforts!

4.1.6 release notes

April 22, 2025

Warning

Upgrading from previous versions

django CMS 4.1 is the **first community release** of django CMS 4. Django CMS 4 introduces changes that **require** action if you are upgrading from a 3.x version. Please read the step-by-step guide to the process of upgrading from 3.5+ to 4 here: *4.0.0 release notes*

Welcome to django CMS 4.1.6!

Django and Python compatibility

django CMS supports **Django 3.2 to 5.2**. We highly recommend and only support the latest release of each series.

It supports **Python 3.9, 3.10, 3.11, and 3.12**. As for Django we highly recommend and only support the latest release of each series.

What's new in 4.1.6

Bug Fixes:

- Django 5.2 tried adding object tools to the page tree throwing an error (#8200) (52af3ffc9) – Fabian Braun

Statistics:

This release includes 4 pull requests, and was created with the help of the following contributors (in alphabetical order):

- Fabian Braun (1 pull request)

With the review help of the following contributors:

- Vinit Kumar

Thanks to all contributors for their efforts!

4.1.5 release notes

April 4, 2025

> **Warning**
>
> Upgrading from previous versions
>
> django CMS 4.1 is the **first community release** of django CMS 4. Django CMS 4 introduces changes that **require** action if you are upgrading from a 3.x version. Please read the step-by-step guide to the process of upgrading from 3.5+ to 4 here: *4.0.0 release notes*

Welcome to django CMS 4.1.5!

Django and Python compatibility

django CMS supports **Django 3.2 to 5.1**. We highly recommend and only support the latest release of each series.

It supports **Python 3.9, 3.10, 3.11, and 3.12**. As for Django we highly recommend and only support the latest release of each series.

What's new in 4.1.5

Bug Fixes:

Bug Fixes:

- Grouper models w/o must not assume language grouper (#8194) (#8195) (35521bc7f) – Fabian Braun
- Ensure correct placeholder retrieval for PageContent instances (#8088) – Fabian Braun
- Fallback page names were not escaped (#8113) (#8114) – Fabian Braun
- Use PageContent.changed_date for sitemap lastmod (#8125) – Jacob Rief
- Allow frontend editing of page title fields – Fabian Braun
- Detect page when getting toolbar for endpoint (#8137) (#8138) – Fabian Braun
- CMS_TOOLBAR_HIDE broke endpoints in django CMS 4+ (#8176) – Fabian Braun
- Preview did not show the redirect page (#8175) – Fabian Braun

Statistics:

This release includes 8 pull requests, and was created with the help of the following contributors (in alphabetical order):

- Fabian Braun (7 pull request)
- Jacob Rief (1 pull request)

With the review help of the following contributors:

- Vinit Kumar
- sourcery-ai[bot]

Thanks to all contributors for their efforts!

4.1.4 release notes

November 12, 2024

Warning

Upgrading from previous versions

django CMS 4.1 is the **first community release** of django CMS 4. Django CMS 4 introduces changes that **require** action if you are upgrading from a 3.x version. Please read the step-by-step guide to the process of upgrading from 3.5+ to 4 here: *4.0.0 release notes*

Welcome to django CMS 4.1.4!

Security fix

django CMS 4.1.4 closes a security vulnerability that could allow an attacker to inject malicious code into the page title allowing to load arbitrary javascript code when viewing the page. We recommend that you upgrade to this version as soon as possible.

The security issue is of low severity, since an attacker needs to have access to the django CMS admin interface to exploit it.

Thanks to Ali İltizar (@alii76tt) for reporting the issue.

Note

As ever, we remind our users and contributors that all security reports, patches and concerns be addressed only to our security team by email, at security@django-cms.org.

Django and Python compatibility

django CMS supports **Django 3.2 to 5.1**. We highly recommend and only support the latest release of each series.

It supports **Python 3.9, 3.10, 3.11, and 3.12**. As for Django we highly recommend and only support the latest release of each series.

What's new in 4.1.4

Bug Fixes:

- XSS vulnerability for page title (#8075) (c045a990e) – Fabian Braun
- Menus crashed when unexpected page content was present (#8052) – Fabian Braun
- Sites menu was empty in the page tree (#8064) – Fabian Braun
- Added redirect message when editing a redirect toolbar object (#8056) – Sal
- X frame options added to page settings form (#8041) – Sal
- template tag `get_admin_url_for_language` did not return the latest page content (#7967) – Fabian Braun
- Sitemap return a QuerySet in CMSSitemap.items() (#8031) – Jens-Erik Weber
- Improved UX when page content is missing in selected language (#8033) – Jacob Rief

Other:

- Updated welcome page (#8057) – Fabian Braun

Statistics:

This release includes 9 pull requests, and was created with the help of the following contributors (in alphabetical order):

- Fabian Braun (5 pull request)
- Jacob Rief (1 pull request)
- Jens-Erik Weber (1 pull request)
- Sal (2 pull request)

With the review help of the following contributors:

- Jacob Rief
- Mark Walker
- Vinit Kumar

Thanks to all contributors for their efforts!

4.1.3 release notes

September 22, 2024

> **Warning**
>
> Upgrading from previous versions
>
> django CMS 4.1 is the **first community release** of django CMS 4. Django CMS 4 introduces changes that **require** action if you are upgrading from a 3.x version. Please read the step-by-step guide to the process of upgrading from 3.5+ to 4 here: *4.0.0 release notes*

Welcome to django CMS 4.1.3!

Django and Python compatibility

django CMS supports **Django 3.2 to 5.1**. We highly recommend and only support the latest release of each series.

It supports **Python 3.9, 3.10, 3.11, and 3.12**. As for Django we highly recommend and only support the latest release of each series.

What's new in 4.1.3

Bug Fixes:

- Respect ContentAdminManager pattern for frontend-editable models (#7998) (e4650ecb7) – Fabian Braun
- Improve pagecontent caching in page admin (esp. page tree) (#8002) (842f347da) – Fabian Braun
- Clear menu cache if page permissions are changed (#7988) (1719b9a1b) – Fabian Braun
- Consistent labels and help texts for page content model and page content forms (#7968) (acbc2e70a) – Fabian Braun

- Inconsistent color codes for dark mode and `prefers-color-scheme: auto` (#7979) (46ff58321) – Fabian Braun
- Invalidate permissions cache if group assignment of user changes (ec05b6f2f) – Fabian Braun
- Accept legacy action names for page permission checks (#8021) (9a1e178) – Fabian Braun
- Mark language and user middleware synchronous for ASGI (#7985) – John Bazik

Statistics:

This release includes 6 pull requests, and was created with the help of the following contributors (in alphabetical order):

- Fabian Braun (6 pull requests)

With the review help of the following contributors:

- Mark Walker
- Vinit Kumar

Thanks to all contributors for their efforts!

4.1.2 release notes

April 20, 2024

> **Warning**
>
> Upgrading from previous versions
>
> django CMS 4.1 is the **first community release** of django CMS 4. Django CMS 4 introduces changes that **require** action if you are upgrading from a 3.x version. Please read the step-by-step guide to the process of upgrading from 3.5+ to 4 here: *4.0.0 release notes*

Welcome to django CMS 4.1.2!

Django and Python compatibility

django CMS supports **Django 3.2 to 5.0**. We highly recommend and only support the latest release of each series.

It supports **Python 3.8, 3.9, 3.10, 3.11, and 3.12**. As for Django we highly recommend and only support the latest release of each series.

What's new in 4.1.2

Improved right-to-left support

- Both, the CSS assets and the JS assets have been adjusted to better support right-to-left languages (RTL).
- Besides imrpvoed ease of use, the page tree can now be fully managed with RTL admin languages
- Improved translations for Arabic

Faster menu rendering

- Improved efficiency building menus for the page tree dramatically reduces database hits
- Page menus rendered by the core both for versioned and unversioned pages

Bug Fixes

- Placeholders do not block deletion of custom model instances with a `PlaceholderRelationField`.
- Structure view respects toolbar language
- Fixed management command to delete orphaned plugins
- Faster DOM update after editing plugins
- Directly redirect to edit endpoint after creating a new page
- Allow editing page content object for apphook without root content
- Render fallback language in place if `redirect_on_fallback` is set to `False` in the `CMS_LANGUAGES` settings
- Fail silently when rendering a placeholder on a missing toolbar object – Fabian Braun
- Show fallback language titles in page tree – Fabian Braun

4.1.1 release notes

April 20, 2024

> **Warning**
>
> Upgrading from previous versions
>
> django CMS 4.1 is the **first community release** of django CMS 4. Django CMS 4 introduces changes that **require** action if you are upgrading from a 3.x version. Please read the step-by-step guide to the process of upgrading from 3.5+ to 4 here: *4.0.0 release notes*

Welcome to django CMS 4.1.1!

Django and Python compatibility

django CMS supports **Django 3.2 to 5.0**. We highly recommend and only support the latest release of each series.

It supports **Python 3.8, 3.9, 3.10, 3.11, and 3.12**. As for Django we highly recommend and only support the latest release of each series.

What's new in 4.1.1

Improved right-to-left support

- Both, the CSS assets and the JS assets have been adjusted to better support right-to-left languages (RTL).
- Besides imrpvoed ease of use, the page tree can now be fully managed with RTL admin languages
- Improved translations for Arabic

Faster menu rendering

- Improved efficiency building menus for the page tree dramatically reduces database hits
- Page menus rendered by the core both for versioned and unversioned pages

Bug Fixes

- Placeholders do not block deletion of custom model instances with a `PlaceholderRelationField`.
- Structure view respects toolbar language
- Fixed management command to delete orphaned plugins
- Faster DOM update after edition plugins
- Directly redirect to edit endpoint after creating a new page
- Allow editing page content object for apphook without root content
- Render fallback language in place if `redirect_on_fallback` is set to `False` in the `CMS_LANGUAGES` settings

4.1.0 release notes

December 20, 2022

> **Warning**
>
> Upgrading from previous versions
>
> django CMS 4.1.0 is the **first community release** of django CMS 4. Django CMS 4 introduces changes that **require** action if you are upgrading from a 3.x version. Please read the step-by-step guide to the process of upgrading from 3.5+ to 4 here: *4.0.0 release notes*

Welcome to django CMS 4.1!

These release notes cover the new features, as well as some backwards incompatible changes you'll want to be aware of when upgrading from django CMS 4.0. If you are upgrading from django CMS 3.11 or earlier please urgently read the release notes of django CMS 4.0.

Django and Python compatibility

django CMS supports **Django 3.2 to 5.0**. We highly recommend and only support the latest release of each series.

It supports **Python 3.8, 3.9, 3.10, and 3.11**. As for Django we highly recommend and only support the latest release of each series.

What's new in 4.1

Status indicators in page tree

- Status indicators are shown in the page tree. For django CMS core only two states are available: public and empty.
- Django CMS core provides hooks to allow other packages to patch the status indicators, e.g., djangocms-versioning.
- Djangocms-versioning will add more functionality to the indicators (e.g., publish from page tree).

Bug Fixes

- In rare cases moving plugins from one placeholder to another could result in a server error and an inconsistent plugin tree.
- Empty page contents (e.g., due to a missing translation of a page) will now render correctly in the page tree.

- Adding a page will trigger the form in the language viewd not in the browser language
- The "Empty all" menus for placeholders now works.

Backward incompatible changes in 4.1

TitleExtension

`TitleExtension` in `cms.extensions.models` has been renamed to `PageContentExtension` to keep a consistent language in the page models.

Any packages using `TitleExtension` will need to adapt the name change in their code base.

In `ExtensionToolbar` the method `get_title_extension_admin(language=None)` has been deprecated. It is recommended to switch to the new `get_page_content_extension_admin(page_content=None)`.

Monkey patching

This is a purely internal change: django CMS v4.1 does not support monkey patching as for djangocms-versioning before version 2.0. Please only use djangocms-versioning >= 2.0

Miscellaneous

- The Django setting `SEND_BROKEN_LINK_EMAILS` (removed from Django since version 1.8) was used as a signal to send an email to the site managers if `page_url` could not reverse the url name. Since this version the outdated setting is ignored. If managers want to receive mails add `django.middleware.common.BrokenLinkEmailsMiddleware` to the project's `settings.MIDDLEWARE`.
- `cms.api.create_title` has been renamed to *`create_page_content`*. A compatibility shim remains and issues a deprecation warning.
- `cms.models.pluginmodel.CMSPlugin.copy_plugin` was removed.

4.0.0 release notes

> **Note**
>
> Version 4.1 is the first **community release of django CMS 4**. It includes all of the changes mentioned in this section and those mentioned in *4.1.0 release notes*. django CMS 3 users seeking to upgrade **should immediately go to version 4.1**.
>
> Version 4.0 has never been released on pypi but is available on github.

This release of django CMS is a complete rewrite of the core, hugely simplifying what django-cms does out of the box. The main reasons for the changes:

- Limitations with publishing, where only 2 versions can ever exist
- Too many "opinions" of how parts of the CMS should work baked in

> **Warning**
>
> Upgrading from previous versions
>
> 4.0.0 introduces some changes that **require** action if you are upgrading from a previous version. Please read below for a step-by-step guide to the process of upgrading from 3.5+ to 4.0.0

The core principles of django-cms 4.0+

- The CMS core such be simplified
- The core should not force any opinions
- The app registry, plugin and wizard pools should be simplified and allow easy registration
- Allow third parties to define how publishing should work, if anyone needs the feature to work differently they can. This is why publishing was moved to djangocms-versioning.

How to upgrade to 4.0.0

It is currently recommended to start new projects on django-cms 4.0.0. The changes from django-cms 3.x to 4.x are so different that only 3rd party utilities can assist with the migration such as *djangocms-4-migration* https://pypi.org/project/djangocms-4-migration/.

Please refer to the guidance within the aforementioned package to perform a migration between projects.

What's new in 4.0.0

New features at a glance

- A revised model structure, delivering huge performance improvements
- Powerful versioning functionality
- A new app configuration facility that allows other apps to customise / control other apps by enhancing features.
- Dedicated Edit, Preview and Structure endpoints. Allows the editing interface to be used by custom models and not just pages.
- New and improved plugin architecture
- New Alias Placeholders that are versioned and provide more control (replaces Static placeholders)

Improvements and new features

Do we comment on changes that have already happened in previous v3 versions, such as Github actions etc??

- Feat Added pre-migrate hook to check version 4 is intentional (https://github.com/django-cms/django-cms/commit/ff6cb9b5dced92eadef62694e989d601e9475b30)
- Feat Added live-url querystring parameter option for PageContent edit and preview endpoints (https://github.com/django-cms/django-cms/commit/ee89fe4f44fb0675bbdb85a2804de5328450a184)
- Fix Structure mode disappearing from the toolbar (https://github.com/django-cms/django-cms/commit/7dafe846a94e50e96e29f0d8909fc25f43cbcaab)
- Fix pagetree and status alignment (https://github.com/django-cms/django-cms/commit/914558d283c197b4035ae7e1a084860f486c9429)
- Feat Upgrade the FE bundle to Node 16 (https://github.com/django-cms/django-cms/commit/f110ddb25083a19263508ccbecfb0c692204245a)
- Feat Allow showing the toolbar for anonymous users (https://github.com/django-cms/django-cms/commit/2008ca8a85eaf5f875d37c2fbca6ce03b2c7b2d8)
- Ported Django 3.2 support (https://github.com/django-cms/django-cms/commit/b0deaedd7d5e11086d10799445b3cd6df47c11a4)
- Ported Django 3.1 support (https://github.com/django-cms/django-cms/commit/fb0d4f235b3b80610356e9a0c89fb361ea5e27c5)

- Ported Django 3.0 support (https://github.com/django-cms/django-cms/commit/c44b6beda941b29cf964c2a2fe28f012d9b6c83f)
- Ported Split database packages so that tests can be run with sqlite (https://github.com/django-cms/django-cms/commit/c77b5e08a1cd2074789cbe461392bc7ac01e11d6)
- Fix being able to reset the setting PageContent.limit_visibility_in_menu (https://github.com/django-cms/django-cms/commit/66c70394c9e144281a0b47d93e3784d06318acf9)
- Ported Replace Travis CI with Github actions (https://github.com/django-cms/django-cms/commit/29ae26eafa0abf4ec27160ba59d890e4497043f6)
- Feat Add CMSAppExtension.ready which is called after all cms app configs are loaded (https://github.com/django-cms/django-cms/commit/c02308fc52610eaeec9ea6b663c89b08614e4317)
- Feat Deprecate the core Alias plugin (https://github.com/django-cms/django-cms/commit/0fec81224889a94bdb7fce6c9f1da2fb7c886ec8)
- Feat Replace deprecated Jquery .load() call with .on('load', ... (https://github.com/django-cms/django-cms/commit/c9cd9fbf26804f74b4df884ae67e8f90603af583)
- Feat Refactor *page.get_title_cache* to be more straightforward (https://github.com/django-cms/django-cms/commit/80911296bba8d263f5150cb481e925cdf307a363)
- Feat Added Prevent JS injection in the admin add plugin url (https://github.com/django-cms/django-cms/commit/72025947d8d3757977f0efab75bda70504a3b6c4)
- Ported Fix 'urls.W001' warning with custom apphook urls (https://github.com/django-cms/django-cms/commit/75978fb1c3ad25d1efba39a5d32215314358ba71)
- Ported Override urlconf_module so that Django system checks don't crash (https://github.com/django-cms/django-cms/commit/f1226a57b767d4b9f66a0cfec4374b5157c49e4e)
- Feat Added raise a 404 when EmptyPageContent is returned (https://github.com/django-cms/django-cms/commit/8e7cdb12d20e63e552ea2cc010f586c3dfbb396a)
- Feat Added the ability to disable the sideframe with the setting *CMS_SIDEFRAME_ENABLED* (https://github.com/django-cms/django-cms/commit/a1ac04d3f81777f6404af62a9c31ff74715b7028)
- Feat Added dedicated edit and preview toolbar buttons (https://github.com/django-cms/django-cms/commit/5005cd933e12332e9296cdda3e0a9eeaea3fc9a2)
- Feat Added expose the sideframe in CMS.API (https://github.com/django-cms/django-cms/commit/4dadf9f1e1f2cf4da6bc68f8367236b040255fbc)
- Feat Removed resolve View (https://github.com/django-cms/django-cms/commit/e3a23a7fc757892c7d58e4af6b78e853ddecab87)
- Feat Removed resolve Page (https://github.com/django-cms/django-cms/commit/0e885ca9e27367c7154cb33406725ac3b67eb170)
- Feat Added toolbar persist setting *CMS_TOOLBAR_URL__PERSIST* (https://github.com/django-cms/django-cms/commit/fb27c34e2a4aebd2e10e1262ef1c43b69c79a132)
- Feat Added front end editing and rendering registry (https://github.com/django-cms/django-cms/commit/db4ff4162cf4ecd36caa8bba066ec28f75b472d8)
- Feat Added Placeholder checks framework (https://github.com/django-cms/django-cms/commit/53171cf2ba7e6aaeca6b2a86df6ad3ffde80e965)
- Feat Registered PageContent with the django admin (https://github.com/django-cms/django-cms/commit/2e090d6c2fd9768f1e8e871dfa9f17ddb2154f7a)

- Feat Added a new source field to PageContent to the Placeholder model (https://github.com/django-cms/django-cms/commit/b075f44d3384b765c74a55947b82ba3c885b0bb1)
- Feat Renamed the Title model to PageContent (https://github.com/django-cms/django-cms/commit/2894ae8bcf92092d947a097499c01ab2bbb0e6df)
- Feat create_page API warning added for no longer accepting a published argument (https://github.com/django-cms/django-cms/commit/f48b8698f239881cc4ca0d593ecae20628486a04)
- Feat Dedicated Edit and Preview endpoints (https://github.com/django-cms/django-cms/commit/bf1af91bf5cc6dba4b19b476201f398cf58e768f, https://github.com/django-cms/django-cms/commit/685361d475fc4718bf0b1e3444a27be8505a7390, https://github.com/django-cms/django-cms/commit/0f12156c8ed85914d4e3b14b30bce87becefe92b, https://github.com/django-cms/django-cms/commit/39562aeb9e61d5d3c08b1031757be11bc5934dff)
- Feat Refactored the plugin tree, replacing django-treebeard with custom CTE queries (https://github.com/django-cms/django-cms/commit/83d38dbb2e51b4cb65aff5726a1c415de7a1c376, https://github.com/django-cms/django-cms/commit/4dfaa1c360c2a15f6572b89fc994a254be9e961d, https://github.com/django-cms/django-cms/commit/90bb064fa794c3cc3decd547dc9ddcc5cb89d100)
- Feat Registered the Placeholder model with the django admin (https://github.com/django-cms/django-cms/commit/5a1c89316f3b58c92910520000d87dbe37b3132a)
- Feat Removed Placeholder content fallbacks (https://github.com/django-cms/django-cms/commit/a9947fed11275bae833d1efdee3e8fa4bc1e0eaf)
- Feat Added Generic Foreignkey to Placeholder model (https://github.com/django-cms/django-cms/commit/0aedfbbd1a1eafb750607a3d0f784fcf118c9532)
- Feat Removed publisher_publish management command (https://github.com/django-cms/django-cms/commit/cb19c60697bbd042b973f7df88f85d2b2a22753b)
- Feat Placeholders moved from Page to the Title model (https://github.com/django-cms/django-cms/commit/37082d074a4e37a9d2114c4236d526529daa1219, https://github.com/django-cms/django-cms/commit/d7e2d26a6c7c6991a8edf2883092ddff6b87c0aa)
- Feat Wizards integrated into the app registration system (https://github.com/django-cms/django-cms/commit/c8f56a969b30b70a8795fc5c15a0aa70b2fe1ae9)
- Feat Page and Placeholder signals rewritten to group Page and Placeholder plugin operations (https://github.com/django-cms/django-cms/commit/03941533670ee9f8c5c078bda8e5cfdd9a639f53, https://github.com/django-cms/django-cms/commit/ca16415b1022c984ce0525336beafacfed14bb31)
- Feat Added new cms app registration and configuration system (https://github.com/django-cms/django-cms/commit/97515c81da2d883055098c0a5c3d033629ea5b15)
- Feat Removed publishing from the core (https://github.com/django-cms/django-cms/commit/41c4ab0dc72e2a3015cd789657924ade09797f0a, https://github.com/django-cms/django-cms/commit/14110d06779399ee90631dc45c21fa419fbeeb9f, https://github.com/django-cms/django-cms/commit/cf442f756f41d0447def9cd2a2bb41d7b8a53cf3, https://github.com/django-cms/django-cms/commit/9905ca6ec986942f3acc692d10deabbc0ca5768d, https://github.com/django-cms/django-cms/commit/1d789468403f50301e413856a925b15f020a71b1, https://github.com/django-cms/django-cms/commit/9f25075455595b11a75ae5574aa4a7ad0c791670)

Bug Fixes

Removal of deprecated functionality

- Removed *Page.get_draft()*
- Removed *Page.get_published()*

- Removed *StaticPlaceholders*

Main differences to django CMS 3.x

The main differences to note in the core CMS which is now extremely simplified are:

- No concept of publishing, removed because it was limited to just draft and live. An opinionated implementation is now accomplished through djangocms_versioning. Many new concepts exist in this application. The reason that the publishing is external is due to the fact that it is an opinionated implementation. If it is agreed as the way forward by the community it could potentially be brought in as an internal app that compliments the core codebase, similar to how Django is organised internally.
- CMS app config, allows other apps to customise / control other apps by enabling or disabling features.
- Dedicated Edit, Preview and Structure endpoints, this allows any applications using Placeholders inside or outside of the CMS (djangocms_alias) to use the same editing experience.
- New plugin architecture, simplified and no reliance on treebeard which was problematic in the past.
- Static placeholders are being replaced by djangocms_alias because static placeholders cannot be versioned or allow moderation.

Model changes

Page, Title (now PageContent) and Placeholder refactor

There are various changes to the model structure for the Page and PageContents (formerly Title). The most notable is the fact that plugins from different Title instances were all saved in the same Placeholder instance. This has now changed in DjangoCMS 4, a PageContent (formerly Title) instance now contains a dedicated set of Placeholder instances.

The model structure was changed to allow flexibility in the core of the cms, this allowed a package such as djangocms-versioning to create infinite PageContent models.

Data model of CMS < 4

- **Page (x1 for Draft and x1 for Live)**
  - Title Language: “EN”
  - Title Language: “DE”
  - Placeholder Slot: “header”
  - **Placeholder Slot: “contents”**
    - Plugin 1 Language “EN”
    - Plugin 2 Language “DE”

Data model of CMS >= 4

- **Page**
  - **PageContents Language: “EN”**
    - Placeholder Slot: “header”
    - **Placeholder Slot: “contents”**
      - Plugin 1 Language “EN”
  - **PageContents Language: “DE”**

        * Placeholder Slot: “header”
        * **Placeholder Slot: “contents”**
            - Plugin 2 Language “DE”

Page, PageContents (Title) and Placeholder relation refactor: https://github.com/django-cms/django-cms/commit/37082d074a4e37a9d2114c4236d526529daa1219

Moving Title to PageContent

The model structure was changed to allow the core of the cms to be flexible and un-opinionated.

To handle the fact that the Title model is renamed in the CMS you will need to import the PageContent model.

For a djangocms 4.0 only project:

```
from cms.models import PageContent
```

For a djangocms 3.x and 4.0 compatible project:

```
# To handle the fact that the Title model is renamed in the CMS you will need to import␣
↪the PageContent model.
try:
    from cms.models import PageContent
# django CMS 3.x
except ImportError:
    from cms.models import Title as PageContent
```

For a djangocms 4.x+ only project:

```
from cms.models import PageContent
```

Settings

New or changed settings added.

CMS_TOOLBAR_ANONYMOUS_ON

default
: `False`

This setting controls if anonymous users can see the CMS toolbar with a login button when `?toolbat_on` is appended to a URL. The default behaviour is to not show the toolbar to anonymous users.

CMS_TOOLBAR_URL__ENABLE

default
: `toolbar_on`

This setting is used to force the toolbar to show on a page.

CMS_TOOLBAR_URL__DISABLE

default
: `toolbar_off`

This setting is used to force the toolbar to be hidden on a page.

App registration

https://github.com/django-cms/django-cms/pull/6421 app registration docs in the description of the PR

- Add-ons now make use of a new config system; this is to be migrated to all pools. Add-ons can now define whether they support other addons (such as versioning) as well as provide configuration. This is useful in telling features like versioning how to version an add-on.
- Previously all add-ons would manage their own pool, now it is moving to an app registry based system that will allow centralised control. Although all new add-ons should implement this system the new system will not be depreciated at this time.
- CMSApp is an existing term from v2.5, it is how apphooks are declared in the newer versions of the cms.
- CMSAPPConfig is a class, which defines the configuration for a specific add-on, this is then passed to CMSAppExtension. It provides a way of telling the core that an app wants to access something from another app config (the centralized way of handling app config). For example: Alias wants to tell versioning to version it. This requires two components, versioning must define CMSAppExtension, all it needs to do is implement one method, called *configure_app*, which takes an instance of the CMSAppConfig. In order for an alias app to be linked to it set *app_name_enabled=True*. When the extension is configured like this the cms will take all the config settings and pass them to the relevant extension, specify models that need to be versioned and which apps need to access this config. CMSAppExtension is the way to register the add-ons and in the future plugins (or plugin_pools) with have their configs defined in CMSAPPConfig.

App configuration example

An application that defines an app extension can be used by other apps by registering as "enabled" in the CMSAppConfig by adding: "package_with_extension_enabled":

```
# A package that defines an app extension for other apps to register with
# myapp/cms_config.py
class MyappCMSExtension(CMSAppExtension):
    def __init__(self):
        self.mylist = []

    def configure_app(self, cms_config):
        if hasattr(cms_config, "myapp_attribute"):
            self.mylist.append(cms_config.myapp_attribute)


# A package that defines a value to add to the extension
# someotherapp/cms_config.py
class SomeotherappCMSConfig(CMSAppConfig):
    # By enabling the someotherapp with myapp, the extension will be used for the␣
↪someotherapp
    myapp_enabled = True
    # Supply a value to `myapp_attribute` to be added to the myapp cms_config.mylist␣
↪attribute.
    myapp_attribute = "A string value"
```

App configuration usage examples in djangocms-url-manager and djangocms-alias

It is configurable in v4 so you can have another Content Type that you want to work with url manager. here is an example of how url does this for the cms Page, shows you the power of the cms config: https://github.com/django-cms/djangocms-url-manager/blob/acffbeedd3950b9d91f971e7a190b2789d2fe9d9/djangocms_url_manager/cms_config.py#L14

If you had a new Content Type and a new application , you can add the config entry in your third party application and url manager would start to use your model.

Here is an example of djangocms-alias configuring itself for versioning: https://github.com/django-cms/djangocms-alias/blob/7d90b7763278ff74ebe49f70420ecb9f0e2dc4c6/djangocms_alias/cms_config.py#L26 versioning knows nothing about Alias, Alias tells versioning how to use it. No more other apps embedding logic. Obviously Page is configured in url manager by default because it depends on django-cms.

Publishing has been moved to djangocms-versioning

- There is no longer the concept of publishing baked into the core of the CMS. By default any content changes are instantly live with no option to unpublish other than to remove altogether.
- To enable publishing the package djangocms-versioning or other similar package that is Django CMS 4.0+ compatible should be installed.
- The reason that publishing was removed from the core is because the solution baked in made a lot of assumptions that enforced various limitations on developers. By not providing a publishing method it allows developers to provide their own solutions to the publishing paradigm.
- Goal is to migrate the monkey patching of versioning into the core to allow a "simple" mode in djangocms-versioning that replaces the 3.x draft/live mode when installing (default option).

See here for the djangocms-versioning documentation.

djangocms-versioning overrides queries from PageContent

- django CMS Versioning overrides the standard query manager for PageContent by adding the query manager: PublishedContentManagerMixin. https://github.com/django-cms/djangocms-versioning/blob/429e50d4de6d14f1088cbdba2be63b20c2885be9/djangocms_versioning/managers.py#L4
- By default only published versions are returned from *PageContents.objects.all()*.

To get all versions regardless of versioning state you can use the "_base_manager":
***PageContent._base_manager.all()*::**

Get only published PageContents PageContent.objects.all()

Get all PageContents regardless of the versioning status, be careful with this as it can return archived, draft and published versions! PageContent._base_manager.all()

Get only draft PageContents from djangcms-versioning.constants import DRAFT PageContent._base_manager.filter(versions__state=DRAFT)

Disabling the admin sideframe

- The CMS sideframe in the Django admin caused many issues when navigating through different plugins admin views, the experience it offered left the user confused at the page they were currently on after making various changes, it was also buggy at times. Disable the sideframe by adding the following setting in the settings.py file, it is enabled by default. CMS_SIDEFRAME_ENABLED = False

Plugin refactor

- Plugins used to utilise Treebeard. The Treebeard implementation was not coping with this, it was prone to breakage and tree corruption. The refactor simplifies and avoids this by utilising a parent child relationship with plugins. The main issue when replacing the Treebeard implementation was performance, here the standard Django ORM could not provide the query complexity and performance required, individual implementations for the different SQL dialects was implemented to aid performance of plugin queries.

- Initial plugin refactor: https://github.com/django-cms/django-cms/commit/83d38dbb2e51b4cb65aff5726a1c415de7a1c376
- Support for other SQL dialects for the plugin tree structure: https://github.com/django-cms/django-cms/commit/4dfaa1c360c2a15f6572b89fc994a254be9e961d

Signals

Page signals have been merged into pre_obj and post_obj signals for operations on Page. Publishing signals have been removed as of DjangoCMS 4.0 but are available in djangocms-versioning: https://github.com/django-cms/django-cms/commit/03941533670ee9f8c5c078bda8e5cfdd9a639f53

Log Operations

Previously the logs created were inconsistent and were not created for all page and placeholder operations. Now all page and placeholder operations are logged in the Django Admin model LogEntry. The logs can also be triggered by external apps via using the signals provided in the CMS. https://github.com/django-cms/django-cms/commit/03941533670ee9f8c5c078bda8e5cfdd9a639f53

Placeholder Admin

The placeholder is now responsible for the edit, structure and preview endpoints. This was previously taken care of by appending *?edit*, *?structure* and *?preview*, This change was made to allow objects that weren't pages to be viewed and edited in their own way (Alias is an example of this).

- The views to render the endpoints: render_object_structure, render_object_edit, render_object_preview located at: https://github.com/django-cms/django-cms/blob/release/4.0.x/cms/views.py#L195 The endpoint is determined by using a reverse look up to the registered admin instance using the toolbar utils: (get_object_preview_url, get_object_structure_url, get_object_edit_url) https://github.com/django-cms/django-cms/blob/release/4.0.x/cms/toolbar/utils.py#L122 This is due to the addition of versioning. Previously every add-on was responsible for their edit end points which made it impossible for versioning to bring the correct end point for a specific version. You need to specify cms_toolbar_enabled_models attribute, which is a list of tuples in the following format: (model, render function). model - model you want to be editable
- render function - a function that takes django.http.HttpRequest object and an object of the model specified above, and returns a django.http.HttpResponse (or any subclass, like TemplateResponse) object based on provided data. Please note that the preview/edit endpoint has changed. Appending ?edit no longer works. There's a separate endpoint for editing (that the toolbar is aware of and links to when clicking Edit button). One also needs to include *cms_enabled* = *True* in the cms config, otherwise that cms_toolbar_enabled_models config won't be passed to the cms.
- PlaceholderAdminMixin is deprecated and has a deprecation notice that it will be removed in the next major release: CMS 5.0. https://github.com/django-cms/django-cms/blob/release/4.0.x/cms/admin/placeholderadmin.py#L178

Placeholder relations

The *PlaceholderField* has been replaced by the *PlaceholderRelationField*, the built-in migrations will automatically take care of the replacement, but it can't however replace the code.

You need to replace your fields such as:

```
class Post(models.Model):
    ...
    media = PlaceholderField("media", related_name="media")
```

with:

```
class Post(models.Model):
    ...
    placeholders = PlaceholderRelationField()
```

The above you may think is very strange, and you are completely correct. This is because the placeholder relationship is now a GenericForeignKey relationship, so it can handle many different placeholders at once.

To be able to use *media* again, we can create a property like the below example:

```
class Post(models.Model):
    ...
    def _get_placeholder_from_slotname(self, slotname):
        try:
            return self.placeholders.get(slot=slotname)
        except Placeholder.DoesNotExist:
            from cms.utils.placeholder import rescan_placeholders_for_obj
            rescan_placeholders_for_obj(self)
            return self.placeholders.get(slot=slotname)

    @cached_property
    def media(self):
        return self._get_placeholder_from_slotname("media")
```

Placeholder endpoints

The Placeholder endpoints are designed in a way that allows other third party packages to reuse the edit and preview modes. The major benefit of the reuse is that a third party package can use the views to manage plugins.

Preview end-point

The preview endpoint replaces what was the *?preview* feature in django-cms 3.x

To generate a preview url you can reuse the following snippet, replacing my_page_content_instance with an instance of PageContent:

```
from cms.toolbar.utils import get_object_preview_url

edit_url = get_object_preview_url(my_page_content_instance)
```

Edit end-point

The edit endpoint replaces what was the *?edit* feature in django-cms 3.x

To generate an edit url you can reuse the following snippet, replacing my_page_content_instance with an instance of PageContent:

```
from cms.toolbar.utils import get_object_edit_url

edit_url = get_object_edit_url(my_page_content_instance)
```

Structure end-point

The structure endpoint is a endpoint used by the plugin sidebar used when viewing the edit endpoint. It's where the plugins are rendered and can be dragged & dropped, added and removed.

Configuring you application to use Placeholder endpoint

We can use djangocms-alias as an example here because this is a very good example of a package that "reuses" the django-cms placeholder endpoints.

Your app should have a placeholder field, djangocms-alias adds this manually. The core CMS has a more advanced technique of adding placeholders by the templates, for django-cms alias we only need one placeholder. Please refer to how the core django-cms package implements this for PageContent if you need more advanced control of Placeholder creation.

It is important that your app uses the concept for djangocms-versioning of a grouper and content model:

```
# models.py

class AliasContent(models.Model):
    ...
    placeholders = PlaceholderRelationField()
    placeholder_slotname = 'content'
```

Within your packages cms_config add the following entry:

```
# cms_config.py

class AliasCMSConfig(CMSAppConfig):
    cms_enabled = True
    cms_toolbar_enabled_models = [(AliasContent, render_alias_content)]
```

Static Placeholders

Static Placeholders have been superseded by djangocms-alias, because they cannot be versioned.

release notes 3.11.5

What's new in 3.11.5

Features:

- feat: Add Python 3.12 support – Vinit Kumar
- feat: django 5 support for cms 3.11 – Leonardo Cavallucci
- feat: Add bot to remind to not squash merges into release/* branches – Fabian Braun

Bug Fixes

- fix: preserve `view_class` in decorated views – Will Hoey
- fix: avoid InvalidCacheKey (memcached) for key-length ~249 – fwehr
- fix: Update transifex pull strings script for v3 in alignment with v4 – Fabian Braun
- fix: Use correct version of Django in GitHub CI actions – Fabian Braun
- fix: Remove link to closed discourse channel for feature requests from issue template – Fabian Braun

- fix: Remove discontinued discourse server from docs – Fabian Braun
- fix: Add `--fix-paths` option to `./manage.py cms fix-tree` – Fabian Braun
- fix: readable messages in dark-mode – Fabian Braun

Statistics:

This release includes 5 pull requests, and was created with the help of the following contributors (in alphabetical order):

- Fabian Braun
- fwehr
- Leonardo Cavallucci
- Vinit Kumar
- Will Hoey

Thanks to all contributors for their efforts!

How to upgrade to

We assume you are upgrading from django CMS 3.11.4.

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 ensure that your database is up-to-date with migrations
python manage.py cms fix-tree
```

Check custom code and third-party applications for use of deprecated or removed functionality or APIs (see above). Some third-party components may need to be updated.

Install the new version of django CMS from GitHub or via pip.

Run:

```
python manage.py migrate
```

to apply the new migrations.

release notes 3.11.4

What's new in 3.11.4

Features:

- Update dark mode switch to be compatible with Django 4.2 admin dark mode (#7549) (1106ae6d7) – Fabian Braun

Bug Fixes:

- Toolbar action button becomes hard to read in dark mode (c626022ba) – Fabian Braun
- Backport v4.1.0rc4 fixes - Admin language and styling (#7630) (#7641) (90b72ebea) – Fabian Braun
- diff-dom freezing on content refresh: #7460 (#7600) (d8e9c527e) – Vinit Kumar
- Fixed RecursionError when extending templates (#7594) (c99f78759) – mihalikv

- JS issues with running CMS under cypress (#7591) (ce4c29948) – Vinit Kumar
- Mitigate performance hit due to deprecation warnings for v4.1 (#7587) (9908d7e70) – Fabian Braun
- create page wizard fails with Asian page titles/unicode slugs (#7565) (0ab640ce3) – Fabian Braun
- require Django >= 3.2 (#7562) (a77358b93) – Fabian Braun
- respect pre-set (48353c2d6) – Fabian Braun
- lint menus app (#7534) (927b60b47) – Vinit Kumar
- remove curly bracket left behind on PR 7488 (#7529) (123f7df91) – Corentin Bettiol

Statistics:

This release includes 45 pull requests, and was created with the help of the following contributors (in alphabetical order):

- ChengDaqi2023 (1 pull request)
- Corentin Bettiol (1 pull request)
- Fabian Braun (30 pull requests)
- Github Release Action (3 pull requests)
- Vinit Kumar (3 pull requests)
- caption (1 pull request)
- dependabot[bot] (0 pull request)
- mihalikv (1 pull request)
- suryadev99 (1 pull request)

Thanks to all contributors for their efforts!

How to upgrade to 3.11.4

We assume you are upgrading from django CMS 3.11.3.

Please make sure that your current database is consistent and in a healthy state, and **make a copy of the database before proceeding further.**

Check your settings of *CMS_LANGUAGES* (if used), as it was ignored by default in preceding versions. For more information, please see: https://github.com/django-cms/django-cms/pull/6795

Then run:

```
python manage.py migrate  # to ensure that your database is up-to-date with migrations
python manage.py cms fix-tree
```

Check custom code and third-party applications for use of deprecated or removed functionality or APIs (see above). Some third-party components may need to be updated.

Install the new version of django CMS from GitHub or via pip.

Run:

```
python manage.py migrate
```

to apply the new migrations.

release notes 3.11.3

This release focuses on Django 4.2 support which will be LTS. django CMS version 3.11.x will be supported until the end of life of Django 4.2 estimated for April 2026.

Compared to 3.11.2 it fixes a bug which broke the dropdown menu in the page tree.

What's new in 3.11.3

Bug Fixes:

- Remove superfluous curly bracket left behind on PR 7488 (#7529) – Corentin Bettiol
- Fix admin tests (#6848) for some post requests (#7535) – Fabian Braun

Statistics:

This release includes 2 pull requests, and was created with the help of the following contributors (in alphabetical order):

- Corentin Bettiol (1 pull request)
- Fabian Braun (1 pull requests)

With the review help of the following contributors:

- Fabian Braun
- Vinit Kumar

Thanks to all contributors for their efforts!

How to upgrade to

We assume you are upgrading from django CMS 3.11.2 or django CMS 3.11.1.

Please make sure that your current database is consistent and in a healthy state, and **make a copy of the database before proceeding further.**

Check your settings of *CMS_LANGUAGES* (if used), as it was ignored by default in preceding versions. For more information, please see: https://github.com/django-cms/django-cms/pull/6795

Then run:

```
python manage.py migrate  # to ensure that your database is up-to-date with migrations
python manage.py cms fix-tree
```

Check custom code and third-party applications for use of deprecated or removed functionality or APIs (see above). Some third-party components may need to be updated.

Install the new version of django CMS from GitHub or via pip.

Run:

```
python manage.py migrate
```

to apply the new migrations.

release notes 3.11.2

This release focuses on Django 4.2 support which will be LTS. django CMS version 3.11.x will be supported until the end of life of Django 4.2 estimated for April 2026.

What's new in 3.11.2

Features:

- add django 4.2 support (#7481) (5478faa5c) – Vinit Kumar
- add setting to redirect slugs to lowercase (#7509) (01aedee9f) – pajowu
- add setting so redirect preserves params (#7489) (dcb9c4b3a) – Ivo Branco
- add download statistics to readme (#7474) (25b2303f7) – Fabian Braun

Bug Fixes:

- replace ‘ by in fr translation no more “page du0027accueil”! (#7488) (b4acc9a6b) – Corentin Bettiol
- Link both user and group from global page permissions to change form (#7486) (6cb47629b) – Fabian Braun
- Build docs always from the current local version (#7472) (#7475) (7aaddd45a) – Fabian Braun

Statistics:

This release includes 21 pull requests, and was created with the help of the following contributors (in alphabetical order):

- Corentin Bettiol (1 pull request)
- Danny Waser (1 pull request)
- Fabian Braun (10 pull requests)
- Ivo Branco (1 pull request)
- Jasper (1 pull request)
- Nihal Rahman (1 pull request)
- Vinit Kumar (3 pull requests)
- pajowu (1 pull request)

With the review help of the following contributors:

- Fabian Braun
- Nihal
- Vinit Kumar

Thanks to all contributors for their efforts!

How to upgrade to

We assume you are upgrading from django CMS 3.11.1.

Please make sure that your current database is consistent and in a healthy state, and **make a copy of the database before proceeding further.**

Check your settings of *CMS_LANGUAGES* (if used), as it was ignored by default in preceding versions. For more information, please see: https://github.com/django-cms/django-cms/pull/6795

Then run:

```
python manage.py migrate  # to ensure that your database is up-to-date with migrations
python manage.py cms fix-tree
```

Check custom code and third-party applications for use of deprecated or removed functionality or APIs (see above). Some third-party components may need to be updated.

Install the new version of django CMS from GitHub or via pip.

Run:

```
python manage.py migrate
```

to apply the new migrations.

3.11.1 release notes

This release focuses on support for django 4 and dark mode.

What's new in 3.11.1

Features:

- add Python 3.11 support for Django CMS (#7422) (3fe1449e6) – Vinit Kumar
- Support for Django 4.1 (#7404) (777864af3) – Fabian Braun
- Add support for tel: and mailto: URIs in Advanced Page Settings redirect field (#7370) (0fd058ed3) – Mark Walker
- Improved dutch translations – Stefan van den Eertwegh

Bug Fixes:

- Prefer titles matching request language (#7144) (06c9a85df) – Micah Denbraver
- Adds a deprecation warning for SEND_BROKEN_LINK_EMAILS (#7420) (d38f4a1cc) – Fabian Braun
- Added deprecation warning to *get_current_language()* (#7410) (2788f75e6) – Mark Walker
- CMS check management command fixed [#7412] (#7413) (dcf394bd5) – ton77v
- Changing color scheme resets session settings to defaults (#7407) (fcfe77f63) – Fabian Braun
- Clear page permission cache on page create (#6866) (e59c179dd) – G3RB3N
- Unlocalize page and node ids when rendering the page tree in the admin (#7188) (9e3c57946) – Marco Bonetti
- Allow partially overriding CMS_CACHE_DURATIONS (#7339) (162ff8dd8) – Qijia Liu
- CMS check management command fixed [#7386] (cdcf260aa) – Marco Bonetti
- default light mode (#7381) (abc6e6c5b) – viliammihalik
- Added language to page cache key (#7354) (d5a9f49e6) – Mark Walker

Refactoring and Cleanups:

- Move js API functions to CMS.Helpers to make them available also to the admin site (#7384) (a7f8cd44f) – Fabian Braun

Statistics:

This release includes 40 pull requests, and was created with the help of the following contributors (in alphabetical order):

- Cage Johnson (1 pull request)
- Christian Clauss (1 pull request)
- Dapo Adedire (1 pull request)
- Fabian Braun (11 pull requests)
- G3RB3N (1 pull request)
- Hussein Srour (1 pull request)
- Marco Bonetti (2 pull requests)
- Mark Walker (10 pull requests)
- Micah Denbraver (1 pull request)
- Qijia Liu (1 pull request)
- Shivan Sivakumaran (1 pull request)
- Vinit Kumar (1 pull request)
- code-review-doctor (1 pull request)
- dependabot[bot] (0 pull request)
- ton77v (1 pull request)
- viliammihalik (1 pull request)
- wesleysima (1 pull request)

With the review help of the following contributors:

- Christian Clauss
- Conrad
- Fabian Braun
- Florian Delizy
- Marco Bonetti
- Mark Walker
- Pankrat
- Patrick Mazulo
- Simon Krull
- Vinit Kumar
- dependabot[bot]
- jefe

Thanks to all contributors for their efforts!

How to upgrade to

We assume you are upgrading from django CMS 3.11.0.

Please make sure that your current database is consistent and in a healthy state, and **make a copy of the database before proceeding further.**

Check your settings of *CMS_LANGUAGES* (if used), as it was ignored by default in preceding versions. For more information, please see: https://github.com/django-cms/django-cms/pull/6795

Then run:

```
python manage.py migrate  # to ensure that your database is up-to-date with migrations
python manage.py cms fix-tree
```

Check custom code and third-party applications for use of deprecated or removed functionality or APIs (see above). Some third-party components may need to be updated.

Install the new version of django CMS from GitHub or via pip.

Run:

```
python manage.py migrate
```

to apply the new migrations.

3.11.0 release notes

This release focuses on support for django 4 and dark mode.

What's new in 3.11.0

Features:

- Add pre commit functionality (#7204) (d1ecb6359) – Mark Walker
- Run workflows in concurrency groups (#7211) (04e843337) – Mark Walker
- Added concurrency option to github workflows (#7205) (546b36827) – Mark Walker
- Add support for django 4 (#7268) (9e8eb17) – Vinit Kumar
- Make Plugin Confirm Template configurable (#7267) (bab1e6e) – Jacob Rief
- Add support for dark mode for toolbar, page tree, structure tree, modals (#7245) (b2d9a08) – Fabian Braun

Bug Fixes:

- release script version number (#7322) (8ffc6488d) – Mark Walker
- add support for custom user model in cms permission signals (#7281) (c10b8ffc3) – Vinit Kumar
- publishing static placeholders outside of CMS (#7253) (bdb50b650) – Adrien Delhorme
- Toolbar bug in 3.10 (#7232) (b12d07989) – Mark Walker
- Disable workflow concurrency to bring stability back to the CI (#7209) (fdad05756) – Mark Walker

How to upgrade to 3.11.0

We assume you are upgrading from django CMS 3.10.0.

Please make sure that your current database is consistent and in a healthy state, and **make a copy of the database before proceeding further.**

Check your settings of *CMS_LANGUAGES* (if used), as it was ignored by default in preceding versions. For more information, please see: https://github.com/django-cms/django-cms/pull/6795

Then run:

```
python manage.py migrate  # to ensure that your database is up-to-date with migrations
python manage.py cms fix-tree
```

Check custom code and third-party applications for use of deprecated or removed functionality or APIs (see above). Some third-party components may need to be updated.

Install the new version of django CMS from GitHub or via pip.

Run:

```
python manage.py migrate
```

to apply the new migrations.

3.10.1 release notes

This release focuses on Python 3.10, node 16 (for build system), and bug fixes.

What's new in 3.10.1

Features:

- Add support for Django 3.2 LTS version
- Page changed_date added to the Page tree admin actions dropdown template #6701 (#7046) (73cbbdb00) – Vladimir Kuvandjiev
- Allow recursive template extending in placeholders (#6564) (fed6fe54d) – Stefan Wehrmeyer
- Added ability to set placeholder global limit on children only (#6847) (18e146495) – G3RB3N
- Replaced Travis.CI with Github Actions (#7000) (0f33b5839) – Vinit Kumar
- Added support for Github Actions based CI.
- Added Support for testing frontend, docs, test and linting in different/parallel CI pipelines.
- Added django-treebeard 4.5.1 support, previously pinned django-treebeard<4.5 to avoid breaking changes introduced
- Improved performance of `cms list plugins` command
- Page changed date added to the Page tree admin actions dropdown

Bug Fixes:

- using .nvmrc to target the right nvm version (3e5227def) – Florian Delizy
- Fixed an issue where the wrong page title was returned (#6466) (3a0c4d26e) – Alexandre Joly
- Add toolbar fix for broken CMS in the release 3.10.x – Vinit Kumar

- Fixed #6413: migrations 0019 and 0020 on multi db setups (#6708) (826d57f0f) – Petr Glotov
- Added fix to migrations to handle multi database routing (#6721) (98658a909) – Michael Anckaert
- Fixed issue where default fallbacks is not used when it's an empty list (#6795) (5d21fa5eb) – Arjan de Pooter
- Fixed prefix_default_language = False redirect behavior (#6851) (34a26bd1b) – Radek Stępień
- Fix not checking slug uniqueness on page move (#6958) (5976d393a) – Iacopo Spalletti
- Fixed DontUsePageAttributeWarning message (#6734) (45383888e) – carmenkow
- Fixed Cache not invalidated when using a PlaceholderField outside the CMS #6912 (#6956) (3ce63d7d3) – Benjamin PIERRE
- Fixed unexpected behavior get_page_from_request (#6974) (#6073) (52f926e0d) – Yuriy Mamaev
- Fixed django treebeard 4.5.1 compatibility (#6988) (eeb86fd70) – Aiky30
- Fixed bad Title.path in Multilanguage sites if parent slug is created or modified (#6968) (6e7b0ae48) – fp4code
- Fixed redirect issues when i18n_patterns had prefix_default_language = False
- Fixed not checking slug uniqueness when moving a page
- Fixed builds on RTD
- Fixed the cache not being invalidated when updating a PlaceholderField in a custom model
- Fixed 66622 bad Title.path in multilingual sites when parent slug is created or modified
- Fixed 6973 bag with unexpected behavior `get_page_from_request`
- Fixed migrations with multiple databases
- Fix styles issues, caused by switching to the `display: flex` on the page tree renderer.
- Fixed missing builtin arguments on main `cms` management command causing it to crash
- Fixed template label nested translation
- Fixed a bug where the fallback page title would be returned instead of the one from the current language
- Fixed an issue when running migrations on a multi database project

How to upgrade to 3.10.1

We assume you are upgrading from django CMS 3.10.0.

Please make sure that your current database is consistent and in a healthy state, and **make a copy of the database before proceeding further.**

Check your settings of *CMS_LANGUAGES* (if used), as it was ignored by default in preceding versions. For more information, please see: https://github.com/django-cms/django-cms/pull/6795

Then run:

```
python manage.py migrate  # to ensure that your database is up-to-date with migrations
python manage.py cms fix-tree
```

Check custom code and third-party applications for use of deprecated or removed functionality or APIs (see above). Some third-party components may need to be updated.

Install the new version of django CMS from GitHub or via pip.

Run:

```
python manage.py migrate
```

to apply the new migrations.

3.10.0 release notes

This release focuses on Python 3.10, node 16 (for build system), and bug fixes.

What's new in 3.10.0

Features:

- Add support for Django 3.2 LTS version
- Page changed_date added to the Page tree admin actions dropdown template #6701 (#7046) (73cbbdb00) – Vladimir Kuvandjiev
- Allow recursive template extending in placeholders (#6564) (fed6fe54d) – Stefan Wehrmeyer
- Added ability to set placeholder global limit on children only (#6847) (18e146495) – G3RB3N
- Replaced Travis.CI with Github Actions (#7000) (0f33b5839) – Vinit Kumar
- Added support for Github Actions based CI.
- Added Support for testing frontend, docs, test and linting in different/parallel CI pipelines.
- Added django-treebeard 4.5.1 support, previously pinned django-treebeard<4.5 to avoid breaking changes introduced
- Improved performance of `cms list plugins` command
- Page changed date added to the Page tree admin actions dropdown

Bug Fixes:

- using .nvmrc to target the right nvm version (3e5227def) – Florian Delizy
- Fixed an issue where the wrong page title was returned (#6466) (3a0c4d26e) – Alexandre Joly
- Add toolbar fix for broken CMS in the release 3.10.x – Vinit Kumar
- Fixed #6413: migrations 0019 and 0020 on multi db setups (#6708) (826d57f0f) – Petr Glotov
- Added fix to migrations to handle multi database routing (#6721) (98658a909) – Michael Anckaert
- Fixed issue where default fallbacks is not used when it's an empty list (#6795) (5d21fa5eb) – Arjan de Pooter
- Fixed prefix_default_language = False redirect behavior (#6851) (34a26bd1b) – Radek Stępień
- Fix not checking slug uniqueness on page move (#6958) (5976d393a) – Iacopo Spalletti
- Fixed DontUsePageAttributeWarning message (#6734) (45383888e) – carmenkow
- Fixed Cache not invalidated when using a PlaceholderField outside the CMS #6912 (#6956) (3ce63d7d3) – Benjamin PIERRE
- Fixed unexpected behavior get_page_from_request (#6974) (#6073) (52f926e0d) – Yuriy Mamaev
- Fixed django treebeard 4.5.1 compatibility (#6988) (eeb86fd70) – Aiky30
- Fixed Bad Title.path in Multilanguage sites if parent slug is created or modified (#6968) (6e7b0ae48) – fp4code
- Fixed redirect issues when i18n_patterns had prefix_default_language = False

- Fixed not checking slug uniqueness when moving a page
- Fixed builds on RTD
- Fixed the cache not being invalidated when updating a PlaceholderField in a custom model
- Fixed 66622 bad Title.path in multilingual sites when parent slug is created or modified
- Fixed 6973 bag with unexpected behavior `get_page_from_request`
- Fixed migrations with multiple databases
- Fix styles issues, caused by switching to the `display: flex` on the page tree renderer.
- Fixed missing builtin arguments on main `cms` management command causing it to crash
- Fixed template label nested translation
- Fixed a bug where the fallback page title would be returned instead of the one from the current language
- Fixed an issue when running migrations on a multi database project

How to upgrade to 3.10.0

We assume you are upgrading from django CMS 3.9.0.

Please make sure that your current database is consistent and in a healthy state, and **make a copy of the database before proceeding further.**

Check your settings of *CMS_LANGUAGES* (if used), as it was ignored by default in preceding versions. For more information, please see: https://github.com/django-cms/django-cms/pull/6795

Then run:

```
python manage.py migrate  # to ensure that your database is up-to-date with migrations
python manage.py cms fix-tree
```

Check custom code and third-party applications for use of deprecated or removed functionality or APIs (see above). Some third-party components may need to be updated.

Install the new version of django CMS from GitHub or via pip.

Run:

```
python manage.py migrate
```

to apply the new migrations.

3.9.0 release notes

This release of django CMS (first community driven release) introduces support for Django 3.2, and bugfix. We tried to catch up with as many long waited feature/bugfix requests as possible.

What's new in 3.9.0

Features:

- Add support for Django 3.2 LTS version
- Page changed_date added to the Page tree admin actions dropdown template #6701 (#7046) (73cbbdb00) – Vladimir Kuvandjiev
- Allow recursive template extending in placeholders (#6564) (fed6fe54d) – Stefan Wehrmeyer

- Added ability to set placeholder global limit on children only (#6847) (18e146495) – G3RB3N
- Replaced Travis.CI with Github Actions (#7000) (0f33b5839) – Vinit Kumar
- Added support for Github Actions based CI.
- Added Support for testing frontend, docs, test and linting in different/parallel CI pipelines.
- Added django-treebeard 4.5.1 support, previously pinned django-treebeard<4.5 to avoid breaking changes introduced
- Improved performance of `cms list plugins` command
- Page changed date added to the Page tree admin actions dropdown

Bug Fixes:

- Fixed an issue where the wrong page title was returned (#6466) (3a0c4d26e) – Alexandre Joly
- Fixed #6413: migrations 0019 and 0020 on multi db setups (#6708) (826d57f0f) – Petr Glotov
- Added fix to migrations to handle multi database routing (#6721) (98658a909) – Michael Anckaert
- Fixed issue where default fallbacks is not used when it's an empty list (#6795) (5d21fa5eb) – Arjan de Pooter
- Fixed prefix_default_language = False redirect behavior (#6851) (34a26bd1b) – Radek Stępień
- Fix not checking slug uniqueness on page move (#6958) (5976d393a) – Iacopo Spalletti
- Fixed DontUsePageAttributeWarning message (#6734) (45383888e) – carmenkow
- Fixed Cache not invalidated when using a PlaceholderField outside the CMS #6912 (#6956) (3ce63d7d3) – Benjamin PIERRE
- Fixed unexpected behavior get_page_from_request (#6974) (#6073) (52f926e0d) – Yuriy Mamaev
- Fixed django treebeard 4.5.1 compatibility (#6988) (eeb86fd70) – Aiky30
- Fixed Bad Title.path in Multilanguage sites if parent slug is created or modified (#6968) (6e7b0ae48) – fp4code
- Fixed redirect issues when i18n_patterns had prefix_default_language = False
- Fixed not checking slug uniqueness when moving a page
- Fixed builds on RTD
- Fixed the cache not being invalidated when updating a PlaceholderField in a custom model
- Fixed 66622 bad Title.path in multilingual sites when parent slug is created or modified
- Fixed 6973 bag with unexpected behavior `get_page_from_request`
- Fixed migrations with multiple databases
- Fix styles issues, caused by switching to the `display: flex` on the page tree renderer.
- Fixed missing builtin arguments on main `cms` management command causing it to crash
- Fixed template label nested translation
- Fixed a bug where the fallback page title would be returned instead of the one from the current language
- Fixed an issue when running migrations on a multi database project

How to upgrade to 3.9.0

We assume you are upgrading from django CMS 3.8.

Please make sure that your current database is consistent and in a healthy state, and **make a copy of the database before proceeding further.**

Check your settings of *CMS_LANGUAGES* (if used), as it was ignored by default in preceding versions. For more information, please see: https://github.com/django-cms/django-cms/pull/6795

Then run:

```
python manage.py migrate  # to ensure that your database is up-to-date with migrations
python manage.py cms fix-tree
```

Check custom code and third-party applications for use of deprecated or removed functionality or APIs (see above). Some third-party components may need to be updated.

Install the new version of django CMS from GitHub or via pip.

Run:

```
python manage.py migrate
```

to apply the new migrations.

3.8.0 release notes

This release of django CMS concentrates on introducing support for Django 3.1 and dropps support for Python 2.7 and 3.4. It also removes support for Django versions below 2.2.

What's new in 3.8.0

Improvements and new features

- Introduced Django 3.1 support.
- Dropped support for Python 2.7 and Python 3.4
- Dropped support for Django < 2.2

Bug Fixes

- Removed `djangocms-column` from the manual installation instructions
- Removed duplicate `attr` declaration from the documentation
- Fixed a reference to a wrong variable in log messages in `utils/conf.py`
- Fixed an issue in `wizards/create.html` where the error message did not use the plural form

How to upgrade to 3.8

We assume you are upgrading from django CMS 3.7.

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 ensure that your database is up-to-date with migrations
python manage.py cms fix-tree
```

Check custom code and third-party applications for use of deprecated or removed functionality or APIs (see above). Some third-party components may need to be updated.

Install the new version of django CMS from GitHub or via pip.

Run:

```
python manage.py migrate
```

to apply the new migrations.

3.7.4 release notes

What's new in 3.7.4

Bug Fixes

- Fixed a security vulnerability in the plugin_type url parameter to insert JavaScript code.

3.7.3 release notes

What's new in 3.7.3

Bug Fixes

- Fixed apphooks config select in Firefox
- Fixed compatibility errors on python 2
- Fixed long page titles in Page tree/list view to prevent horizontal scrolling

3.7.2 release notes

What's new in 3.7.2

Bug Fixes

- migrated from `django.utils.six` to the six package
- migrated from `django.utils.lru_cache` to `functools.lru_cache`
- migrated from `render_to_response` to `render` in `cms.views`
- added `cms.utils.compat.dj.available_attrs`
- added `--force-color` and `--skip-checks` in base commands when using Django 3
- replaced `staticfiles` and `admin_static` with `static`
- replaced djangocms-helper with django-app-helper

Improvements and new features

- Added support for Django 3.0
- Added support for Python 3.8

How to upgrade to 3.7.2

Django 3.0 changed the default behaviour of the `XFrameOptionsMiddleware` from `SAMEORIGIN` to `DENY`. In order for django CMS to function, `X_FRAME_OPTIONS` needs to be set to `SAMEORIGIN` in the `settings.py`:

```
X_FRAME_OPTIONS = 'SAMEORIGIN'
```

3.7.1 release notes

What's new in 3.7.1

Bug Fixes

- Fixed a bug where creating a page via the `cms.api.create_page` ignores left/right positions.
- Fixed documentation example for `urls.py` when using multiple languages.
- Fixed a bug where `request.current_page` would always be the public page, regardless of the toolbar status (draft / live). This only affected custom urls from an apphook.
- Fixed a bug where the menu would render draft pages even if the page on the request was a public page. This happens when a user without change permissions requests edit mode.
- Fixed the 'urls.W001' warning with custom apphook urls
- Fixed missing `{% trans %}` to toolbar shortcuts.
- Fixed a simple typo in the docstring for `cms.utils.helpers.normalize_name`.

Improvements and new features

- Added code of conduct reference file to the root directory
- Moved contributing file to the root directory
- Added better templates for new issue requests
- Mark public static placeholder dirty when published.
- Prevent non-staff users to login with the django CMS toolbar
- Improved and simplified permissions documentation.
- Improved apphooks documentation.
- Improved CMSPluginBase documentation.
- Improved documentation related to nested plugins.
- Updated installation tutorial.
- Updated branch and release policy.

3.7.0 release notes

This release of django CMS concentrates on introducing support for Django 2.2 LTS and Python 3.7.

What's new in 3.7.0

Improvements and new features

- Introduced Django 2.2 support.

- Introduced Python 3.7 support.
- Fixed test suite.
- Fixed override `urlconf_module` so that Django system checks don't crash.

How to upgrade to 3.7

We assume you are upgrading from django CMS 3.6.

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 ensure that your database is up-to-date with migrations
python manage.py cms fix-tree
```

Check custom code and third-party applications for use of deprecated or removed functionality or APIs (see above). Some third-party components may need to be updated.

Install the new version of django CMS from GitHub or via pip.

Run:

```
python manage.py migrate
```

to apply the new migrations.

Create a new django CMS 3.7 project

On the Divio Cloud

The Divio Cloud offers an easy way to set up django CMS projects. In the Divio Cloud Control Panel, create a new django CMS project and **Deploy** it.

Using the django CMS Installer

> **Note**
>
> The django CMS Installer is not yet available for django CMS 3.6 or Django 2 or later.
>
> This section will be updated or removed before the final release of django CMS 3.6.

Contributors to this release

- Daniele Procida
- Vadim Sikora
- Paulo Alvarado
- Bartosz Płóciennik
- Katie McLaughlin
- Krzysztof Socha
- Mateusz Kamycki

- Sergey Fedoseev
- Aliaksei Urbanski
- heppstux
- Chematronix
- Frank
- Jacob Rief
- Julz
- Angelo Dini

3.6.1 release notes

What's new in 3.6.1

Bug Fixes

- Fixed a security vulnerability in the plugin_type url parameter to insert JavaScript code.

3.6.0 release notes

This release of django CMS concentrates on introducing support for Django 2.0 and Django 2.1, and dropping support for Django versions lower than 1.11.

What's new in 3.6.0

Improvements and new features

- introduced support for Django 2.0
- introduced support for Django 2.1
- removed support for Django versions older than 1.11
- added `page_title` parameter for `cms.api.create_page()` and `cms.api.create_title()`
- length restriction for `Title.meta_description` was moved from model to form; field length was increased to 320 characters.

Removal of deprecated functionality

Previously deprecated functionality has been removed:

- Signal handlers for `Page`, `Title`, `Placeholder` and `CMSPlugin` models was removed.
- Removed the `cms moderator` command.
- Removed the translatable content `get/set` methods from `CMSPlugin` model.

How to upgrade to 3.6

We assume you are upgrading from django CMS 3.5.

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 ensure that your database is up-to-date with migrations
python manage.py cms fix-tree
```

Check custom code and third-party applications for use of deprecated or removed functionality or APIs (see above). Some third-party components may need to be updated.

Install the new version of django CMS from GitHub or via pip.

Run:

```
python manage.py migrate
```

to apply the new migrations.

Create a new django CMS 3.6 project

On the Divio Cloud

The Divio Cloud offers an easy way to set up django CMS projects. In the Divio Cloud Control Panel, create a new django CMS project and **Deploy** it.

Using the django CMS Installer

> **Note**
>
> The django CMS Installer is not yet available for django CMS 3.6 or Django 2 or later.
>
> This section will be updated or removed before the final release of django CMS 3.6.

Contributors to this release

- Daniele Procida
- Vadim Sikora
- Paulo Alvarado
- Bartosz Płóciennik
- Katie McLaughlin
- Krzysztof Socha
- Mateusz Kamycki
- Sergey Fedoseev
- Aliaksei Urbanski
- heppstux
- Chematronix
- Frank
- Jacob Rief
- Julz

3.5.4 release notes

What's new in 3.5.4

Bug Fixes

- Fixed a security vulnerability in the plugin_type url parameter to insert JavaScript code.

3.5.3 release notes

What's new in 3.5.3

Bug Fixes

- Fixed `TreeNode.DoesNotExist` exception raised when exporting and loading database contents via `dumpdata` and `loaddata`.
- Fixed a bug where `request.current_page` would always be the public page, regardless of the toolbar status (draft / live). This only affected custom urls from an apphook.
- Removed extra quotation mark from the sideframe button template
- Fixed a bug where structureboard tried to preload markup when using legacy renderer
- Fixed a bug where updates on other tab are not correctly propagated if the operation was to move a plugin in the top level of same placeholder
- 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 plugins that are not rendered in content wouldn't be editable in structure board
- Fixed a bug with expanding static placeholder by clicking on "Expand All" button
- Fixed a bug where descendant pages with a custom url would lose the overwritten url on save.
- Fixed a bug where setting the `on_delete` option on `PlaceholderField` and `PageField` fields would be ignored.
- Fixed a bug when deleting a modal from changelist inside a modal

3.5.2 release notes

What's new in 3.5.2

Bug Fixes

- Fixed a bug where short-cuts menu entry would stop working after toolbar reload
- Fixed a race condition in frontend code that could lead to sideframe being opened with blank page
- Fixed a bug where the direct children of the homepage would get a leading `/` character when the homepage was moved or published.
- Fixed a bug where non-staff user would be able to open empty structure board
- Fixed a bug where a static file from Django admin was referenced that no longer existed in Django 1.9 and up.
- Fixed a bug where the migration 0018 would fail under certain databases.

3.5.1 release notes

What's new in 3.5.1

Bug Fixes

- Fixed a bug where editing pages with primary keys greater than 999 would throw an exception.
- Fixed a `MultipleObjectsReturned` exception raised on the page types migration with multiple page types per site.
- Fixed a bug which prevented toolbar js from working correctly when rendered before toolbar.
- Fixed a bug where CMS would incorrectly highlight plugin content when plugin contains invisible elements
- Fixed a regression where templates which inherit from a template using an `{% extends %}` tag with a default would raise an exception.

3.5.0 release notes

This release of django CMS concentrates on usability and user-experience, by improving its responsiveness while performing editing operations, particularly those that involve updates to plugin trees.

It also continues our move to decouple logical layers in the system. Most significant in this release is the new separation of the structure board from page rendering, which allows the structure board to be updated without requiring the page to be re-rendered. This vastly speeds up page editing, especially when dealing with complex plugin structures.

Another significant example is that the Page model has been decoupled from the site navigation hierarchy. The navigation tree now exists independently, offering further speed advantages, as well as future benefits for development and extensibility.

Our work to improve separation of concerns can also be seen in the renaming of publishing controls, so that they no longer refer to specifically to pages. Ultimately, publishing actions could apply to any kind of content, and this is a step in that direction.

What's new in 3.5.0

Improvements and new features

- structure board now decoupled from page rendering
- `Page` model decoupled from the site navigation
- Page copy between sites
- better behaviour of the language chooser for published/unpublished languages
- improved handling, refactored code for language fallbacks
- improved repr for `Page`, `Title`, `Placeholder` and `CMSPlugin` models
- generic publishing controls no longer refer to "page"
- improved documentation

Bug Fixes

This release fixes:

- a Page template settings permission issue (failed to check for "Change advanced settings permission")
- a bug allowing Pages to be pasted without the correct translations for the target site
- a bug that prevented users from seeing the welcome screen when debug is off

- a bug allowing aliased plugins to render even if their host page was unpublished
- a bug where focusing inputs in modal would require two clicks in some browsers
- minor issues with initialisation of interface widgets.
- minor clipboard bugs

Removal of deprecated functionality

Previously deprecated functionality has been removed:

- Menu modules can no longer be named `menus.py` (use `cms_menus.py`).
- The `cms.utils.django_load.py` module has been removed (in favour of standard Django helpers)
- Support for Django Reversion has been removed.
- The `urls` and `menus` attributes are no longer supported on `CMSApp` (apphook) classes. All apphook subclasses now need a `get_urls()` method. In addition, if your apphook has a `menus` attribute, that will need to be replaced by a `get_menus()` method.
- `Page.revision_id` has been removed
- Deprecated content creation wizard settings have been removed.

Backward-incompatible changes

- The home page is no longer automatically the root page in the tree (since there is no longer a page tree). Instead, the home page is set manually in the page list admin.
- Previously, ordered pages could be obtained via `Page.object.order_by('path')`; the equivalent is now `Page.object.order_by('node__path')`.
- Pages are no longer ordered by path. For ordering, use `order_by('node__path')`.
- Pages no longer have a `site` field. Whereas previously you could use `filter(site=id)`, now use `filter(node__site==id)`.
- Pages no longer have a `parent` field. Instead a `parent` property now returns the new `parent_page` attribute, which relies on the node tree.
- Never-published pages can no longer have a 'pending' publishing state. A data migration, `cms/migrations/0018_pagenode.py`, removes this.
- Using `self.request.path` or `self.request.path_info` in a `CMSToolbar` subclass method is no longer reliable and is discouraged. Instead, use `self.toolbar.request_path`.

How to upgrade to 3.5

We assume you are upgrading from django CMS 3.4.

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 ensure that your database is up-to-date with migrations
python manage.py cms fix-tree
```

Check custom code and third-party applications for use of deprecated or removed functionality or APIs (see above). Some third-party components may need to be updated.

Install the new version of django CMS from GitHub.

Run:

```
python manage.py migrate
```

to apply the new migrations.

Create a new django CMS 3.5 project

On the Divio Cloud

The Divio Cloud offers an easy way to set up django CMS projects. In the Divio Cloud Control Panel, create a new django CMS project and **Deploy** it.

Using the django CMS Installer

See our installation guide in the tutorial. However, make sure that you:

- have installed the latest version of django CMS Installer (at least version `0.9.8`)
- specify the version to install as `develop`: `djangocms --cms-version=develop mysite`

The user name and password will both be `admin`.

Contributors to this release

- Alexander Paramonov
- Andras Gyömrey
- Daniele Procida
- Gianluca Guarini
- Iacopo Spalletti
- Jacob Rief
- Jens Diemer
- Júlio R. Lucchese
- Leon Smith
- Ludwig Hähne
- Mark Walker
- Nicolas PASCAL
- Nina Zakharenko
- Paulo Alvarado
- Robert Stein
- Salmanul Farzy
- Sergey Fedoseev
- Shaun Brady
- Stefan Foulis
- Tim Graham

- Vadim Sikora
- alskgj

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

Improvements and new features

Deprecations

Backward incompatible changes

Page methods

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 favour 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 short-cuts 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 short-cuts 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 short-cuts
- 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 favour 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 behaviour 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 *Contributing 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.

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

Plugin data migrations require care when keeping a project compatible with both django CMS 3.0 and 3.1 — see the django CMS 3.1 release notes archives for the details.

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 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:

Managing permissions on apphooks

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/django-cms/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/django-cms/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/django-cms/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/django-cms/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/django-cms/djangocms-video

Link Plugin

We removed the link plugin (`cms.plugins.link`). Its new location is at:

- https://github.com/django-cms/djangocms-link

Snippet Plugin

We removed the snippet plugin (`cms.plugins.snippet`). Its new location is at:

- https://github.com/django-cms/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
```

(continues on next page)

(continued from previous page)

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

admin.autodiscover()

urlpatterns = i18n_patterns('',
    re_path(r'^admin/', include(admin.site.urls)),
    re_path(r'^', include('cms.urls')),
```

(continues on next page)

(continued from previous page)

```
)

if settings.DEBUG:
    urlpatterns = patterns('',
    re_path(r'^media/(?P<path>.*)$', 'django.views.static.serve',
            {'document_root': settings.MEDIA_ROOT, 'show_indexes': True}),
    re_path(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'),
                'fallbacks': ['de', 'fr'],
                'public': True,
                'hide_untranslated': True,
                'redirect_on_fallback':False,
            },
            {
                'code': 'de',
                'name': gettext('Deutsch'),
                'fallbacks': ['en', 'fr'],
                'public': True,
```

(continues on next page)

(continued from previous page)

```
            },
            {
                '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>
      {% 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('',
    re_path(r'^media/(?P<path>.*)$', 'django.views.static.serve',
            {'document_root': settings.MEDIA_ROOT, 'show_indexes': True}),
    re_path(r'', include('django.contrib.staticfiles.urls')),
) + urlpatterns
```

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

```
python manage.py collectstatic
```

5.3.6 Contribute

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

Contribute to django CMS

As an open source project, django CMS is only as strong as its community. Without the donation of time and skill of our contributors and the financial support of our association members it would not be possible to maintain the django CMS project. The community is the backbone of django CMS.

Our contributors come from all over the world and have different levels of skills and expertise. No matter if you are a developer, usability enthusiast, designer or copywriter. Young or old. Experienced or inexperienced. Every contribution, however small, is valued.

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.

Open source contribution can include taking part in discussions, filing bug reports, proposing improvements, contributing code or writing documentation.

Remember that contributions to the documentation are highly prized, and key to the success of the django CMS project.

All activity in the community is governed by our *Code of Conduct*.

3 Reasons Why You Should Contribute

1. Boost your reputation

Through your involvement as a contributor, other people become aware of your work. In this way, you make a name for yourself in the community and your reputation grows. This can also help you in your professional career. Add your contribution to your resume or LinkedIn profile.

2. Find a mentor and improve your skills

When you join a workgroup, you will receive guidance and support from the workgroup leader. Our working group leaders are professionals in their field and often in leading positions in their respective companies. Take advantage of this unique opportunity for personal development!

3. Meet new people and increase your network

Let's face it: Ultimately, it's the people who bring the django CMS project to life and fill it with joy. Through your involvement, you'll meet new people and maybe even make new friends. As a community, it's important to us to create a pleasant atmosphere where everyone feels welcome!

Are you new to django CMS?

If you are new to django CMS, then we recommend you to first familiarize yourself with the CMS and start with the install section. After that, you can have a go at issues on Github that are marked Good first issue. These issues are especially good if you're just starting out but still want to contribute.

Contributor Community

But before you start getting your hands dirty, you should make sure to join us online in order to stay updated with the latest news and to connect with other users across the world.

You can join us online through our support channels

You should make sure to join our Discord server. It is our main communication platform. Users from all over the world use Discord to talk about django CMS and to support each other in answering support requests. StackOverflow is a very popular, community-based space to find and contribute answers to technical challenges

You can also follow:

- the django CMS Youtube account
- the django CMS Association LinkedIn account

How to contribute

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/django-cms/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.
4. After that, please join our Discord server (#contributors). This group of friendly community members is dedicated to reviewing pull requests. Report your PR and find a "pr review buddy" who is going to review your pull request.
5. Get acknowledged by the django CMS community for your contribution

See *Contributing a patch* 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 Discord 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. New code must follow the ruff formatter (and the pre-commit hook will ensure this). A few highlights:

- **Line Length**: Default is 119 characters.
- **Indentation**: 4 spaces per indentation level; not more, not less.
- **Quotes**: We use double quotes (").
- **Trailing Commas**: Included where syntactically valid, such as in function arguments and list literals.

- **Function Definitions**: Parameters are each on their own line if the function signature spans multiple lines.
- **Docstrings**: Single-line docstrings use double quotes and are formatted with consistent indentation.
- **Blank Lines**: Two blank lines before top-level definitions; one blank line between methods inside a class.
- **Imports**: Grouped and ordered: standard library, third-party, then local imports. Each group is separated by a blank line.

HTML

- **Naming**: Use clear, meaningful names.
- **Indentation**: 4 spaces, no tabs.
- **IDs vs Classes**: Prefer classes.
- **Modularity**: Create reusable components.

CSS

- **Naming**: Follow BEM (Block Element Modifier).
- **Nesting**: Keep Sass nesting shallow (1–2 levels).
- **Formatting**: - 4 spaces indentation. - One selector per line. - Lowercase properties/values (unless uppercase is needed).
- **Ordering**: 1. Positioning 2. Box model 3. Typography 4. Visual 5. Miscellaneous

JavaScript

- **Naming**: - *camelCase* for variables/functions - *PascalCase* for classes - *UPPER_CASE* for constants
- **Formatting**: - 4 spaces indentation - Always use semicolons - Use single quotes (unlike Python quotes!)
- **Implementation**: - Wrap code in IIFEs - Use *'use strict';*
- **Patterns**: Prefer modular, reusable patterns.

JS Linting

JavaScript is linted using ESLint. In order to run the linter you need to do this:

```
npx 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 a Discord Server and of course the code reviews mechanism on GitHub - do use them.

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 18.19.0 (will install npm 10.2.3 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:

```
npx gulp sass
```

While developing it is also possible to run a watcher that compiles Sass files on change:

```
npx gulp watch
```

By default, source maps are not included in the compiled files. In order to turn them on while developing just add the `--debug` option:

```
npx 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:

```
npx 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 a patch

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:

```
cd django-cms
python3 -m venv .venv
source .venv/bin/activate
```

4. Install its dependencies:

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

   Replace `X.Y` with whichever version of Django you want to work with. Check the supported versions in the "test_requirements/" directory

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.

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(admin.ModelAdmin, metaclass=CMSPluginBaseMetaclass):

    ...

    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.test_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:

```
Found 0 test(s).
System check identified no issues (0 silenced).

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

NO TESTS RAN
```

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:

```
Found 1 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.
```

(continues on next page)

(continued from previous page)

```
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
Destroying test database for alias 'default'...
```

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.get_placeholders(language='en')
        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:

```
Found 1 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.
----------------------------------------------------------------------
Ran 1 test in 0.018s

OK
Destroying test database for alias 'default'...
```

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 language
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.

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 American 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.

Documentation requirements

The packages required by the documentation are managed by pip-tools, which compiles `requirements.txt` ensuring compatibility between packages.

The packages that the documentation requires are in `requirements.in` which looks like a regular requirements file. Specific versions of packages can be specified, or left without a version in which case the latest version which is compatible with the other packages will be used.

Example `requirements.in`:

```
furo
Sphinx>4
sphinx-copybutton
sphinxext-opengraph
sphinxcontrib-spelling
pyenchant>3
```

By running `pip-compile` the requirements are compiled into `requirements.txt`.

Periodically requirements should be updated to ensure that new versions, most importantly security patches, are used. This is done using the `-U` flag:

```
cd docs
pip-compile -U
```

The generated `requirements.txt` pins specific versions and explains where each required package comes from, for example:

```
datetime==4.3
    # via -r requirements.in
django==3.2.5
    # via
    #   django-classy-tags
    #   django-cms
    #   django-formtools
    #   django-sekizai
    #   django-treebeard
django-classy-tags==2.0.0
    # via
    #   django-cms
    #   django-sekizai
django-cms==3.9.0
    # via -r requirements.in
django-formtools==2.3
    # via django-cms
```

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

Prefer American English spelling, as it is the standard used in Python, Django, JavaScript, CSS and Django CMS.

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:

- /introduction/index (`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
- *Explanation* (`explanation`): explanations of key parts of the system
- *Reference* (`reference`): technical reference for APIs, key models and so on
- *Contribute* (`contributing`)
- *Release notes & upgrade information* (`upgrade`)
- *Who is behind django CMS* (`who`): who is behind the django CMS project

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 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.

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

Review

All patches should be made as pull requests **against develop-4** 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 one member of the core team who has expertise in the relevant areas, and excluding the author of the pull request.

Proposal and discussion of significant changes

New features and backward-incompatible changes should follow the best practice of DEPS and should be discussed in the community first. After your proposal has been reviewed by the community, it needs to be finally approved by the Tech Committee. 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.

So before submitting pull requests with significant changes, please make sure that the community agrees and the Technical Committee approves.

To create a proposal…

1. please use this DEP template

2. create a discussion in the main Github repository
3. discuss, discuss, discuss
4. join the Tech Committee (#technical-committee) and make the team aware of your proposal after the proposal has been reviewed by the Technical Committee, it is put to a vote at one of the weekly meetings of the technical committee

Release schedule

The roadmap can be found on our website. The release schedule is managed by the release management workgroup. The plan is to release quarterly and according to a retrospective approach.

Example of retrospective approach.

- Q1 2021 -> 3.9 Release
- End of Q1 2021 -> freeze
- Check what's available
- Merge in anything that's been approved
- Q2 2021 Release -> 3.10
- ...
- Unscheduled Releases -> e.g. bug fix -> 3.x.x

Release management is managed on Discord in the #technical-committee channel. For questions regarding the release process please join the channel and reach out. We're happy to help.

Long-Term Support Release

For the current Long-Term Support (LTS) release overview see *here*. *Long-term support* means that this version will continue to receive security and other critical updates in alignment with the corresponding Django LTS 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

We maintain a number of branches on our GitHub repository:

`-main`
: The default target branch for on-going development and new pull requests.

`release/x.y.z` are the latest released versions of django CMS. Commits
: are cherry-picked from `main` and merged into `release/x.y.z` when suitable. We **officially support** the latest, highest released version and the latest LTS.

Please always open PR's against `main` and indicate that they should be backported to the latest LTS release when necessary. Older branches are not supported any longer.

Commits

Commit messages

We follow the Conventional Commits specification for commit messages. Pull requests are linted against this specification so please make your PR title match the specification.

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

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.

Guidelines for django CMS projects

> **Note**
>
> These guidelines are based on the best practice established by the Jazzband project, a community of contributors that shares the responsibility of maintaining Python-based projects.

The django CMS ecosystem consists of many custom projects. Often these projects are maintained by the author themselves. However, sometimes it can make sense to put a project in the care of the django CMS project. Either because it is of interest to the entire community, or because the author can no longer devote time to maintain the project themselves.

Whether an existing project is transferred to the django CMS Github organization, or a new project is set up within the django CMS Github organization, it is important that certain standards are followed.

Acceptance criteria for new projects or existing ones

Projects must meet the criteria of viability, documentation, testing, code of conducts and contributing guidelines. But before that, they must be approved by the Tech Committee.

Approval by Tech Committee of the django CMS Association

New projects or project transfers under the django CMS patronage must first be approved by the Tech Committee. For that you should join the #tech-committee channel on Discord and simply submit your proposal. Then, the TC decides whether or not your project is in line with the product roadmap and overall vision for django CMS.

Viability

Projects to be maintained by the django CMS project must have a certain maturity (No proof of concepts, one-off toys or code snippet hosts) and provide useful functionality. They should also be transferred to django CMS with the agreement of the previous maintainer and in consultation with the Tech Committee (see Tech Committee).

Documentation

Project documentation is one of the most important aspects of a project. For this reason, it is of utmost importance that the project includes prose documentation for end users and contributors. It is also strongly recommended to prepare inline code documentation, as this is considered an indicator of high quality code. Please document as much as possible,

but also as clearly and concisely as possible. To quote Jazzband "Write like you're addressing yourself in a few years." More information about how to contribute software documentation can be found here.

Tests

Your contributions and fixes are more than welcome as are your tests. We do not want to compromise our codebase. Therefore, you are going to have to include tests if you want to contribute. For more information about running and writing tests please see here.

Conduct

Projects are required to adopt and follow the django CMS code of conduct. Please see the Contributor Code of Conduct for more information about what that entails and how to report conduct violations.

Contributing Guidelines

Projects have to add a CONTRIBUTING.md (Markdown) or CONTRIBUTING.rst (reStructuredText) file to their repository so it's automatically displayed when new issues and pull requests are created.

The respective file needs to contain this text:

First of all, thank you for wanting to contribute to the django CMS. We always welcome contributions, like many other open-source projects. We are very thankful to the many present, past and future contributors, to our community heroes and to the [members of the django CMS Association. This is a django CMS project. By contributing you agree to abide by the Contributor Code of Conduct and follow the guidelines. Of course extending the contributing document with your project's contributing guide is highly encouraged, too. See GitHub's documentation on contributing guidelines for more information.

Move an existing project to the django CMS Github organization

To initiate the transfer to django CMS, you should use Github's Transfer Feature to transfer the repository to the django CMS organization.

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 in any 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 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 https://github.com/django-cms/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,
# 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 report any issues on our GitHub repository.

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.

ERROR: zlib is required unless explicitly disabled using --disable-zlib, aborting

If you run into that issue, make sure to install zlib using Homebrew:

```
brew install libjpeg zlib && brew link --force zlib
```

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.

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 lint`. In order to run linting continuously, do:

```
gulp watch
```

Unit tests

Unit tests can be run like this:

```
gulp unitTest
```

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 unitTest --tests=cms.base,cms.modal
```

If you want to run tests continuously you can use the watch command:

```
gulp unitTest --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.

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.

See how to write a test patch.

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 given a warning, and if necessary will be excluded from participation in official django CMS channels 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 us via info@django-cms.org and our Community Manager will reach out to you.

Your concerns will be taken seriously, treated as confidential and investigated. You will be informed, in writing and as promptly as possible, of the outcome.

5.3.7 Who is behind django CMS

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.

the django CMS Association

In July 2020 Divio handed over the banner to the newly founded django CMS Association (dCA). Its goal is to drive the success of django CMS, by increasing customer happiness, market share and open-source-contributions. Divio remains thoroughly committed to django CMS as the host of the django CMS project website and as one of the founding members of the dCA, next to What. and Eliga Services.

The dCA's role in steering the project's development is formalised in the django CMS technical committee, whose members are drawn from the django CMS community and the dCA.

The dCA 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, the dCA reserves the right of final say in any decisions concerning its development.

As a non-profit organization the django CMS Association is dependent on donations to fulfill its mission, which is based on the following three statements:

- Innovate and lead
- Foster contribution
- Drive adoption

The best way to donate is to become a member of the association and pay membership fees. The funding is funneled back into core development and community projects.

- Sign up for more information about becoming a member of the dCA

The dCA Tech Committee

Mission

It prepares and updates the technical roadmap for approval by the Executive Board and/or the General Assembly, manages incoming feature requests and proposals and takes decisions on awarding credits for work submitted by members.

- Find out more about the mission

Team

- Overview of the team

Tasks

- Tasks & Decisions Log
- Kanban Board

Processes

- Become a core contributor
- Become a member of the Tech Committee

PYTHON MODULE INDEX

C

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W